METAMORPHIC TESTING AND MUTATION TESTING FOR SVM IMAGE CLASSIFICATION

A Thesis-Equivalent Project

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfilment
of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska at Omaha

by

Alex Borchers

May, 2023

Supervisory Committee:

Harvey Siy, Ph.D.

Myoungkyu Song, Ph.D.

Xin Zhong, Ph.D.

Chun-Hua Tsai, Ph.D.

METAMORPHIC TESTING AND MUTATION TESTING FOR SVM IMAGE CLASSIFICATION

Alex Borchers, M. S.

University of Nebraska, 2023

Advisor: Harvey Siy, Ph.D.

The objective of this research project is to investigate the use of metamorphic, mutation and differential testing for machine learning applications. Metamorphic relations are transformations applied to known test inputs which enable one to anticipate the effect on outputs. Mutation testing are operations that change the application behavior to identify gaps in test inputs. Differential testing compares the test outputs of multiple similar applications to determine unexpected behavior. To develop a proof of concept, these tests were carried out on a Support Vector Machine (SVM) applied to image classification tasks. They were implemented on top of an existing online interactive image classification application. The extended application allows the user to build their own model with the selection of different metamorphic transformations and hyperparameters. The model is then trained on a set of images that the user classifies interactively. Once the model is trained, the user can generate hyperparameter mutations via a ChatGPT API, and analyze the impact that different hyperparameters changes have on the confidence and accuracy of the model. This application is designed to serve as an educational and exploratory tool for ML engineers and data scientists. It provides an opportunity for users to gain an understanding of metamorphic relations, various hyperparameters, and various post-training mutations, and their potential impact on ML models.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Machine Learning (ML) allows software applications to make intelligent decisions on their own based on previously recorded data. The system is typically trained using some set of data and some framework, which allows it to create logic to predict future outcomes. ML has many applications in the real world, including the development of self-driving cars, air traffic control systems, and automated medical diagnosis, among countless others. The demand for ML-based applications continues to rise, and with it, the need for software engineering techniques that support reliable development of such systems. This project will investigate how techniques originally developed for software testing can be adapted to test ML applications.

Software testing is an important part of any software system as it helps detect and prevent bugs, and ultimately increases the robustness and reliability of the system. ML testing is important for the same reasons. It helps protect the data that is input into the system, as well as ensure that the output from the ML algorithm is working as it is intended to. Testing ML systems can be difficult due to there being no test oracle for validating the quality that the model is being trained on. For these reasons, ML testing must be approached with novel techniques that do not fit the mold of traditional software testing.

Many techniques have been devised to test systems with no oracle [24]. *Metamorphic testing* is a technique where new test cases are derived from existing test inputs [4]. Utilizing some properties of the system under test, an inference is made to determine the expected output produced using the derived inputs. Thus, just as the derived inputs are related to the original inputs, the expected outputs from the derived inputs are also related to the original

outputs. These relations are known as *metamorphic relations*. Metamorphic testing has become a common technique for testing systems with no known oracles such as search engines, autonomous systems, and ML systems.

Mutation testing [3, 14] and differential testing [15] are also widely used techniques used in software testing. *Mutation testing* is a fault-based software testing technique that introduces bugs (mutations) to a program that represent mistakes a programmer often makes. This is used to identify weak software tests or to find weakly tested pieces of code. Similarly, differential testing is interested in identifying defects in code as well. *Differential testing* is a technique for comparing the behavior of two versions of an application on the same test inputs. Both techniques can be adopted into ML testing to help ensure the correctness and robustness of ML models.

This project focuses on creating a tool to explore what it means to test an ML model using metamorphic testing, mutation testing, and differential testing. The application developed offers a no-code solution to select and visualize metamorphic relations, select hyperparameters, label images, and ultimately train multiple models on a specific set of data. This provides insight into the machine learning process, insight into various ML testing techniques, and novel approaches for hyperparameter mutation.

This report provides a review of recent literature related to ML testing, the experiments we conducted on ML testing techniques, and the application developed as proof of concept, and future work for new researchers seeking involvement in this area.

Chapter 2
Background and Related Work

2.1 Testing Emphasis

2.1.1 Identifying Implementation Bugs in Machine Learning Based Image Classifiers using Metamorphic Testing

Typical verification techniques used in software testing that involve developing test inputs and target inputs are in-feasible for Machine Learning. The leading reasons deemed by Dwarakanath et al. [10] are 1) the program can be expected to take a large number of inputs (i.e. an image classifier), 2) the expected output is unknown in many cases, and 3) finding one or a few cases of incorrect classifications from ML does not indicate the presence of a bug.

The authors propose the use of metamorphic testing to evaluate a machine learning algorithm for implementation bugs. Metamorphic testing is a prominent approach to testing a Machine Learning application. How this type of testing works by building a test case that has two tuples of (input1-output1) and (input2-output2). The second input (input 2) is created such that the tester can reason about the relationship between output1 and output2. That is, input1 should be modified in some way to create an input2 which should hold the relationship from input to output regardless of how the Machine Learning Algorithm trains and classifies the data. A simple example of this is mirroring input1 to create input2. If all inputs are modified in this way, the tester would expect the classification and confidence levels to remain the same for the second tuple.

There were two separate applications tested in this study. The first tested digit recognition using SVM, and the second tested image classification using ResNet. For each image classifier, four metamorphic relationships were developed to test the software for implementation bugs. The authors validated this approach by injecting bugs into the software using mutations generated from Python's MutPy library. The results show that all mutated bugs were caught during the SVM experiments, and 50 percent were caught for ResNet. The authors proved the potential use of metamorphic relations for software testing and left future work to assess other deep learning architectures using a similar approach.

2.1.2 FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds

Randomness in Machine Learning Algorithms impacts how developers write tests that check for the quality of their implementations. In particular, selecting thresholds for comparing quality metrics obtained from these algorithms is a non-intuitive task. This can lead to flaky test results, where a test may fail or not meet some threshold but would not be considered an incorrect implementation.

The authors [9] present a novel tool to fix this problem. FLEX, the first tool of its kind, is used to automatically fix flaky tests caused by algorithmic randomness. This tool helps to approximate assertion values to compare expected and actual values that help represent the quality of the ML software. Dutta et al's technique utilizes Peak Over Threshold method from Extreme Value Theory statistics which identifies an acceptable threshold value to minimize flakiness. FLEX runs many executions of a given test and analyzes the tail distribution of the values returned, looking for a light tail convergence, which can be used to generate a threshold to a given confidence level. From 35 tests collected from 21 real-world ML projects, FLEX identifies and proposes a fix for 28 tests.

2.1.3 Audee: Automated Testing for Deep Learning Frameworks

The quality of Deep Learning systems has become crucial, especially for areas that are safety-critical, such as self-driving vehicles. Existing work focuses mainly on the quality of the models, but lacks when evaluating the underlying framework that these models rely on. Guo et al. [12] propose Audee, a novel approach to automatically test Deep Learning Frameworks. This tool adopts a search-based approach and implements 3 different mutation strategies to generate diverse test cases that explore combinations of model structures, parameters, weights, and inputs of the deep learning frameworks.

Audee diversifies each test by 1) using different layers within the DNN, 2) using different parameters within a layer, 3) using different inputs within the DNN, and 4) using different weights. Each mutation is done on the input level, the weight level, or both for each layer of the DNN. The tool is able to detect three types of bugs: logical bugs, crashes, and NaN errors. In addition, the authors use a cross-reference check to detect inconsistencies across multiple frameworks. The authors validate this approach by testing against 7 DNNs (LeNet, ResNet, VGG, MobileNet, SimpleRNN, LSTM, and GRU) and 4 DL frameworks (TensorFlow, PyTorch, CNTK, and Theano). Audee was able to identify 26 previously unknown bugs, 7 of which had been confirmed or fixed by developers at the time of the study.

2.1.4 Detecting Numerical Bugs in Neural Network Architectures

There are various approaches to test or verify DL models, but none address 1) architecture vendors who design and publish neural architectures to be used by other users, and 2) developers who use neural architectures to train and deploy a model based on the developer's own training dataset. The authors of this study propose a static analysis approach for detecting numerical bugs in Neural network architectures which would help address these

needs. The main contributions of this paper include DEBAR, a static analysis tool used for numerical bug detection in neural architectures, five abstraction techniques provided for arithmetic abstractions and tensor abstraction, and an evaluation of this tool against 9 known buggy architectures and 48 real-world architectures.

Zhang et al. [26] used an abstraction technique to generalize and speed up the analysis of the neural networks. For arithmetic computation abstraction, they used interval abstraction, array expansion, and array smashing. For tensor abstraction, the authors used tensor partitioning and affine relation analysis. The study shows promising results for the use of such a tool in the real world, with future work being left to specially design abstraction techniques for improving scalability and accuracy of detecting numerical bugs.

2.1.5 Multiple-Implementation Testing of Supervised Learning Software

Srisakaokul et al. [22] introduce a multiple-implementation testing strategy to help improve the results from popular machine learning implementations and algorithms. The results from each strategy are evaluated to find what the authors define as the majority-voted output. Any tests that deviate from the majority-voted output (up to some pre-set threshold) are deemed as failing tests. The final output from the tool is a report which highlights the potentially buggy implementations along with areas of the code to evaluate. This approach was able to detect 13 real faults and 1 potential fault against 19 k-Nearest Neighbor algorithms and was able to detect 7 real faults and 1 potential fault against three popular open-source ML projects (Weka, RapidMiner, and KNIME).

2.1.6 CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries

Many developers using Deep Learning libraries assume that the libraries are working as intended and focus on testing the results that are output from these models. Pham et al. [20] take a closer look at how to evaluate, find, and localize bugs in Deep Learning software libraries. The authors propose a tool called CRADLE (Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries) which helps to complete these tasks. CRADLE uses cross-implementation inconsistency checking to detect bugs in DL Libraries and looks for spikes in the execution graph to localize bugs.

The cross-implementation inconsistency checking is done by comparing the results from multiple state-of-the-art DL libraries such as TensorFlow, CNTK, and Theano. The results from each execution are compared against each other using two distance-based metrics (Class-based distance and MAD-distance). Based on the results of the metrics, a pair of backends is deemed inconsistent if a certain percent of the validations instances is larger than a given threshold. Once an inconsistency is found, the tool looks for spikes in the execution to see where the two back-ends deviated from each other (this is done due to the randomness at each layer of a neural network that causes minor deviations when comparing multiple back-ends). This method was tested against 11 public datasets and 30 DL models, from which 104 unique inconsistencies were found.

2.1.7 Testing Probabilistic Programming Systems

Probabilistic systems help automate various parts of common inference tasks and support many approximation algorithms in Machine Learning and Statistics. The authors of this study evaluate the feasibility of using such systems to test ML models for the existence of bugs. Dutta et al. [8] created an application called ProbFuzz which takes a user-defined

input and outputs a set of probabilistic programs which are used to reveal likely bugs. The tool has three main components: the generator, the translator, and the program checker. The generator completes hols in the pre-made template to produce a probabilistic program and the data necessary to run the program. The translator converts the program to the language of the system under test. The templates are written in a high-level probabilistic language notation (IR) which allows for ease of transfer to other languages. Last is the program checker, which runs the generated programs and determines whether the outputs likely indicate a bug. The tool is capable of checking for crashes, NaN and overflow, performance, differential testing with exact and approximate results, and accuracy comparisons.

ProbFuzz was validated by testing against three probabilistic systems: Edward, Pyro, and Stan. Note: Edward uses TensorFlow and Pyro uses PyTorch, two popular ML frameworks seen in other studies. In total, 67 new bugs were discovered by the tool, and 10 were rediscovered. The authors evaluated each bug manually and sent the results to the developers for each of the probabilistic programming systems. 51 were accepted, 8 were rejected, 7 were pending, and 1 had already been fixed at the time of this study. The new bugs found suggest that ProbFuzz is effective at finding bugs in ML frameworks.

2.1.8 Perception and Practices of Differential Testing

Differential Testing is a technique used to analyze and identify differences in the behavior of machine learning models. The authors of this study [11] investigate how ML developers perceive and use differential testing in practice. To do this, they conducted surveys and interviews with developers working with ML models, reviewing how each individual approaches differential testing. The findings revealed that many developers were aware of the importance of this type of testing, but faced challenges in effectively implementing the strategy. Some challenges included a lack of standardization, difficulty generating diverse test sets, and a general lack of understanding of how to evaluate the results of differential

testing.

The paper highlights the importance of effective testing practices to ensure the reliability and fairness of ML models. This strategy for testing ML models is valuable because it can help identify biases and other issues that may affect the performance of machine learning models in real-world scenarios. The paper reveals the need for continued research and development in the field of ML testing and highlights the importance of effective testing practices for building reliable and trustworthy ML systems.

2.2 Robustness Emphasis

2.2.1 Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach

The robustness of neural networks to adversarial examples has received great attention due to the security and high-risk implications or incorrect implementation. The robustness calls into question the safety-critical applications deployed by neural networks, such as autonomous driving systems, and malware detection protocols, among others. Current methods to increase robustness include adversarial training, which is a technique where adversarial examples are generated and the network is retrained to correct labels and improve against these specifically identified examples. The problem is that even such adversarial examples leave the model open to unseen attacks.

The authors [23] propose a metric called CLEVER, which stands for Cross Lipschitz Extreme Value for Network Robustness. This is the first metric that is attack-independent and can be applied to any arbitrary neural network classifier, and that scales to large networks for ImageNet. Weng et al. derive a universal lower bound required to craft adversarial examples. Using this, they prove that the lower bound associates with the maximum norm of the local gradients with respect to the original example, and therefore

the question of robustness becomes that of a Lipschitz Constant problem. To calculate the Lipschitz constant, they use Extreme Value Theory to generate a number that fulfills provides a solution to this problem to some user-defined confidence threshold.

2.2.2 DeepMutation++: a Mutation Testing Framework for Deep Learning Systems

The current state of DNN testing is still in its early stages, with many challenges ahead. This study [13] looks at ways to push forward the field of DNN testing by introducing bugs into code through mutation. The two areas of DNN analyzed here are feed-forward neural networks (FNN) and recurrent neural networks (RNN). Hu et al. created a tool called DeepMutation++ which uses three steps to evaluate these models for vulnerability and robustness. Step 1 is identifying the model under analysis (FNN/RNN). Step 2 is the mutant generation, which is done using a variety of mutation operators specific to each network. Step 3 is vulnerability and robustness analysis, which analyzes the differences between the original DNN and the generated mutants against test inputs for robustness. This analysis is output in a report that indicates the test data quality and the DNN robustness at the same time.

2.3 Synthesis Matrix

Table 2.1: Synthesis Matrix (page 1)

Idea (X) Research (Y)	Dataset Types	Bug Localization and Detection	Statistical Techniques Used
FLEX [9]	Variety, unsure without looking more closely at each.	FLEX evaluates tests repetitvely against an assertation value. Any value generated that is less than (or greater than) the value is deemed flaky and produces a fix.	Extreme Value Theory (to find statistically confident bounds on thresholds for ML programs).
Audee [12]	Image classification and text-based analysis	Layer-Based Casual Testing (determines which layers and which parameters are the source of the bugs detected). Layer change rate (the measure of output change of a certain layer compared to the adjacent previous layers) is used to localize bugs.	NA
DeepMutation++ [13]	Image classification and text-based sentiment analysis	NA - Tool is used to test vulnerability of program against mutations.	”KScores” (1 and 2) used as ”killing score metrics” to approximate vulnerability.
Detecting Numerical Bugs in NN [26]	NA (Neural Network architectures)	Abstracted values that caused errors in static analysis were identified as buggy lines.	None explicitly used to generate thresholds, but many formula’s used in abstraction and measuring accuracy.
Metamorphic Testing [10]	Image Classification	Idenfitication of any metamorphic relations that did not hold determined if a bug existed.	NA
CRADLE [20]	Image Classification	Localize the error spikes that propogate through the execution graph.	Class-Based distance and Mean Absolute Deviation (MAD)-based distance
Multiple-Implementation Testing of Supervised Learning [22]	Image classification and text-based analysis (focus on data used to test classification algorithms)	Bugs are localized manually. Each deviating test is traced back to find faults in the program.	K Nearest Neighbor and Naive Bayes algorithm.
Testing Probabilistic Systems [8]	NA (Probabilistic Systems)	The tool checks for crashes, NaN, overflow, performance, differential testing, and accuracy comparisons.	SMAPE (Symmetric Mean Percentage Error) is used to test accuracy of the new programs.
Evaluating Robustness of NN: EVT Approach [23]	Image Classification	NA	Extreme Value Theory and Lipschitz Continuity.

Table 2.2: Synthesis Matrix (page 2)

Idea (X) Research (Y)	Validation Approach	Analysis of Results
FLEX [9]	The authors tested flaky tests in 35 github projects. From these projects, FLEX generated 28 potential fixes. 19 pull requests were submitted, 9 were accepted, 4 were pending at the time of the paper, and 6 were rejected. 9 remained unsubmitted.	Use of EVT (Extreme Value Theory) to give statistical confidence that a value is unlikely to be above or below some threshold. The algorithm checks for convergence to some heavy or exponential tail. This provides a new assertion value with statistical confidence to some user identified percentage.
Audee [12]	Audee was tested against 7 different DNN's. Each network was tested with the 4 combinations of TensorFlow, CNTK, Pytorch, and Theano. 4 different mutation strategies were used.	The authors use an inconsistency fitness function, which helps to determine how layers of two functions differ from one another.
DeepMutation++ [13]	The authors evaluate the robustness of models in two use cases of DeepMutation ++. The first of which (table 3) shows that the number of mutants tends to increase as they increase the mutation ration. The 2nd measures the robustness of RNN models (LSTM and GRU).	50 test inputs are mutated 100 times to obtain 100 prediction results. KScore2 is calculated against each one to measure robustness.
Detecting Numerical Bugs in NN [26]	Two data sets were used to test their method. The first was a set of 9 architectures that were known to have bugs (they used these to validate their approach worked). The second data set was 48 architectures from a large colllection of research projects in the repository of TensorFlow Research Model.	The tool found all errors in the 9 architectures. From the second data set, 299 previously unknown operations that may lead to numerical errors were found. The tool correctly classified 3,073 operations, with only 230 false positives (93% accuracy). All architectures were analyzed within an average of 12 seconds.
Metamorphic Testing [10]	Results were tested for an existing SVM application and ResNet application.	For SVM, the application was able to catch all of the 12 mutants, while MRs defined for ResNet applications were able to catch 8 of 16 mutants. Total results showed 20/28 mutants were caught or 71%.
CRADLE [20]	Class-Based distance was measured accross 11 public datasets and 30 DL models using 2 different DL architectures/models (Xception and NASNetLarge) and using TensorFlow and CNTK on Keras 2.2.2 and 2.2.1.	104 unique inconsistencies were found.
Multiple-Implementation Testing of Supervised Learning [22]	This was tested against 23 open-source projects (Weka, RapidMiner, KNIME). From these 23 open source projects, 19 implementations of the k-nearest neighbor, and 7 implementations of the Naive Bayes algorithm were tested.	The results are listed in Table 1. In summary, 20.5% of the tests evaluated were deviating tests, with 97.5% revealing a fault.
Testing Probablistic Systems [8]	The authors validated their approaches based on testing on Edward, Pyro, and Stan.	67 new bugs were discovered by ProbFuzz and 10 were 'rediscovered'.

Chapter 3
ML Testing Experiments on Jupyter Notebooks

In this chapter, we explain various experiments conducted in this research. We used Jupyter Notebooks as the primary platform for conducting experiments with ML testing. These experiments enabled us to:

1. conduct ML training and analysis comprehensively on a large dataset, which would take too long to execute within our web-based testing environment, and
2. incrementally test and prototype certain features before integrating into our web-based testing environment.

3.1 Data Source

We utilized a curated dataset consisting of 18,000 images of maize plants from a field trial habitat [25]. Each image has been labelled as healthy or afflicted with northern leaf blight. The original dataset is partitioned into handheld, boom, and drone datasets. The handheld dataset images are captured using a handheld camera. The next two datasets – the boom dataset and the drone dataset were captured using top-mounted cameras in the field and a drone respectively. We utilized the handheld images in our study.

3.2 Using Bitmaps as Input Data

While most conventional SVM image classification models extract image features, e.g., color separation, image intensities, textural features, etc., to form feature vectors as inputs,

we decided to use the actual image bitmap as the model input. This removes the need for processing the image for feature extraction. We believe this would make it easier to utilize the same ML testing infrastructure for other deep learning models like CNN which also commonly take bitmap inputs. Using this led to the model input files having 28,800 features (the number of RGB pixels in a 120x80-sized image). The original images were scaled from 6,000x4,000 to 120x80 for performance improvement. This is explained in Section 4.8.

3.3 Creating Metamorphic Relations

The research began by attempting to recreate the metamorphic relations defined by Dwarakanath et al. [10] which included swapping images RGB channels, rotating an image, mirroring an image, normalizing an image, and scaling the image by a constant. To do this, we first stored all of the target images in a local directory. We then iterated through all of the files and transformed the files into bitmaps that could be used for training and test data suitable for an ML model. We were able to use the Python library, Pillow [6], to transform images to pixels as well as scale the images down. Once the images were scaled, we applied a variety of image transformations:

- RGB Channel: Re-organizing the RGB channels of an image. To do this, we first split the bitmap into three channels using image.split. This splits the bitmap into three sections (r, g, b) according to the channels of an image. We can use Pillow's Image.merge, which allows us to take three lists of the same size and merge them into a new image.

Figure 3.1: RGB Channel (RGB to GBR)

- Multiply by Constant: Multiplying each bitmap value by some constant. This may range from 0.5 to 16 in our application but should hold for any constant. We can directly multiply any list by a constant and it will make this modification.

Figure 3.2: Multiply By Constant (x2)

- Transform: Rotating the image 90, 180, 270 degrees and/or mirroring the image. Numpy arrays have built-in features for rotating and transposing (mirroring) images. We used this to accomplish this relationship.

Figure 3.3: Transform (mirroring)

- Normalize: In this application, we normalized the data by dividing each bitmap value by the standard deviation of the bitmap. This is similar to multiplying by a constant and a similar approach was used for this relation. The picture becomes nearly unrecognizable due to the change, but the relative values among pixels remains intact.

Figure 3.4: Normalize

- Invert: Inverting all pixels in a bitmap. This is a direct mapping for 0 to 255, 1 to 254,...,254 to 1, 255 to 0. This can be done by iterating through all values in the bitmap and subtracting the bitmap value from 255.

Figure 3.5: Inverted

- Permute: A random permutation of the pixels (as long as the permutation is consistent across all images). To do this, we create a mapping of original indexes to new indexes, then iterate through this mapping, which produces our new set of images.

Figure 3.6: Permute

All these input transformations utilized the same overarching metamorphic relation, that is, the transformed inputs will produce the same classification output as using the original inputs. This is because in the bitmap input used as training data with each pixel being one input vector, each vector is independent of the others. Thus even the case where we have

permuted the image should still lead to the same trained model where the support vectors will have isomorphic relationships with the support vectors of the original trained model.

3.3.1 Results

To validate the metamorphic relations, we ran tests for each of the relations mentioned above, training multiple models (one with the original data set and one or more with the modified data set) and compared how they classified the test set. Each of the relations mentioned above is validated using a separate notebook, and they all followed the same framework for validation. All models were trained with an 80-20 split (80 percent training, 20 percent testing) over 1,787 images found in images_handheld_resized folder. For an example, the RGB channel relation compared the results from six different machine learning models (all using different configurations of the RGB channel). Each model was trained using the same images, the only difference being the modifications made to the data. The model was also tested using the same set of images. The classifications from all six models were directly compared, all returning the same exact classifications.

3.4 Determining Optimal Hyperparameters Using GridSearchCV

One dimension that we wanted to include in the application was the comparison of the user's trained model against a pre-trained model. This leads us to a search to find the best-trained model for this set of data. Hyperparameter selection plays a major role in the confidence and accuracy of the ML model. We used GridSearchCV [21] to help find the optimal hyperparameters for this model. GridSearchCV takes as input an ML algorithm (SVM in this case) and a parameter grid. The parameter grid is the option that the developer would like to explore using in the application. GridSearchCV then trains the model against a set of training data and compares it to a set of test data for all possible combinations provided

in the parameter grid. Each combination is trained on five different folds of data (random splits of training/test data). The parameters that produce the highest average across the five folds are deemed as the best estimators.

3.4.1 Results

We first defined the parameter grid with all available options for the kernel ($poly$, $linear$, RBF, $sigmoid$) as well as a range of values for C and $gamma$. These were chosen because they are typically the most influential hyperparameters for ML image classification. From this test, $poly$ and $linear$ were deemed as the highest-performing kernels. We then repeated the results individually for $poly$ and $linear$, with more parameter options specific to each of the kernels. For $poly$, we further defined values for $degree$, $coef0$, and $gamma$. For $linear$, we further defined values for $loss$, $maxiteration$, and C. Similar to the MR validation, each model was trained with an 80-20 split (80 percent training, 20 percent testing) over 1,787 images found in images_handheld_resized folder. After all testing, we deemed $kernel = poly$, $C = 0.1$, and $gamma = 0.0001$ to be the best hyperparameter options, with the kernel-specific values to have little impact on the performance of the ML model.

3.5 Saving the Optimal Model in a Pickle File

Once the optimal hyperparameters were found, we stored the file in a Pickle format for use in the application [2]. While more standard file formats like ONNX or HDF5 could have been used [7], we just needed a simple way to save the model and load it in another Python program. Pickle is available via the Python library sklearn, and can easily transform any trained machine learning model into a file that can be accessed at runtime of an application. This model was trained using the hyperparameters mentioned above, and loaded as a pretrained model by our application (see Section 4.4).

Chapter 4
Project Application

The application was inspired by Dwarakanath et al's research on Metamorphic Testing for identifying bugs in ML-based image classifiers [10]. We sought to build upon an existing application developed in previous research by an undergraduate capstone team at UNO [17]. This application was designed to identify images of maize leaves and classify whether future images were healthy or not based on user selection. The purpose of the application was to help farmers build confidence in the potential uses of ML for classifying leaves.

We extended the application to include capabilities for metamorphic testing, mutation testing, and differential testing [1]. This tool is now meant to be an exploratory tool that can be used to learn about Metamorphic Testing, and as a platform to test robustness, quality of images for ML training, and the impact of hyperparameter and support vector mutation. The target user are data scientists and ML engineers looking for new ways to test ML models and as a potential framework for future engineers to test new sets of data. In this section, we will cover the main contributions to the application throughout this past year of work and research.

4.1 Converting to SVM and Bitmap Inputs

We decided to follow Dwarakanath et al's initial analysis, which used Support Vector Machine (SVM) and bitmap inputs [10]. The existing application used an image preprocessor to extract feature vectors for input to a Random Forest Classifier. Therefore, our first goal was to convert this application to a SVM so that we could apply metamorphic testing as proposed by Dwarakanath et al. The first step to complete this task included replacing

all instances of RandomForestClassifier() in the current application with the new SVM approach and replacing the image inputs from feature vectors into bitmaps.

Since RandomForestClassifier and SVM are both ML algorithms available via sklearn, replacing these instances was straightforward. The use of bitmap inputs (explained in Section 3.2) was more complicated. The RandomForestClassifier accepted a set of feature vectors, consisting of extracted image features, as the training and test data. This drastically changed the shape of the input data, moving from 15 features to approximately 28,800 features (the number of pixels in a 120x80-sized image). This transition forced us to gather the training and test data during a user's session because CSV files could not hold the amount of data needed. This presented major challenges later on that we will discuss.

4.2 Implementing Metamorphic Relation in Application

This implementation follows 3.3. For this part of the development we were focused on three aspects: 1) Allowing the user to select which Metamorphic Relation that they wanted to apply to their images, 2) Transforming the images on screen so the user could see the changes visually, and 3) Training a 2nd ML model alongside the original so we could compare the results from each.

The first and second aspects lead to the development of menu.html (the first route after index.html). This page allows users to select different metamorphic relations together and see what impact these have on the images (see Figure 4.1). An event listener is applied to all input elements on the first page of menu.html, on change the image refreshes with the user's new selections. The user's preferences are saved as session variables once the user moves to the labeling process.

The third takes into consideration the user's selections that are stored in the session variable. The function getData goes through all available images in our data set and transforms that into bitmaps. Once each image is transformed, a second set of data is

generated simultaneously. Using a new class (ImageMR), the system has methods to handle all of the metamorphic relations defined in section 3.1. The two ML models (original and modified) are trained using the same class (createMLModel) but using these two separate data objects.

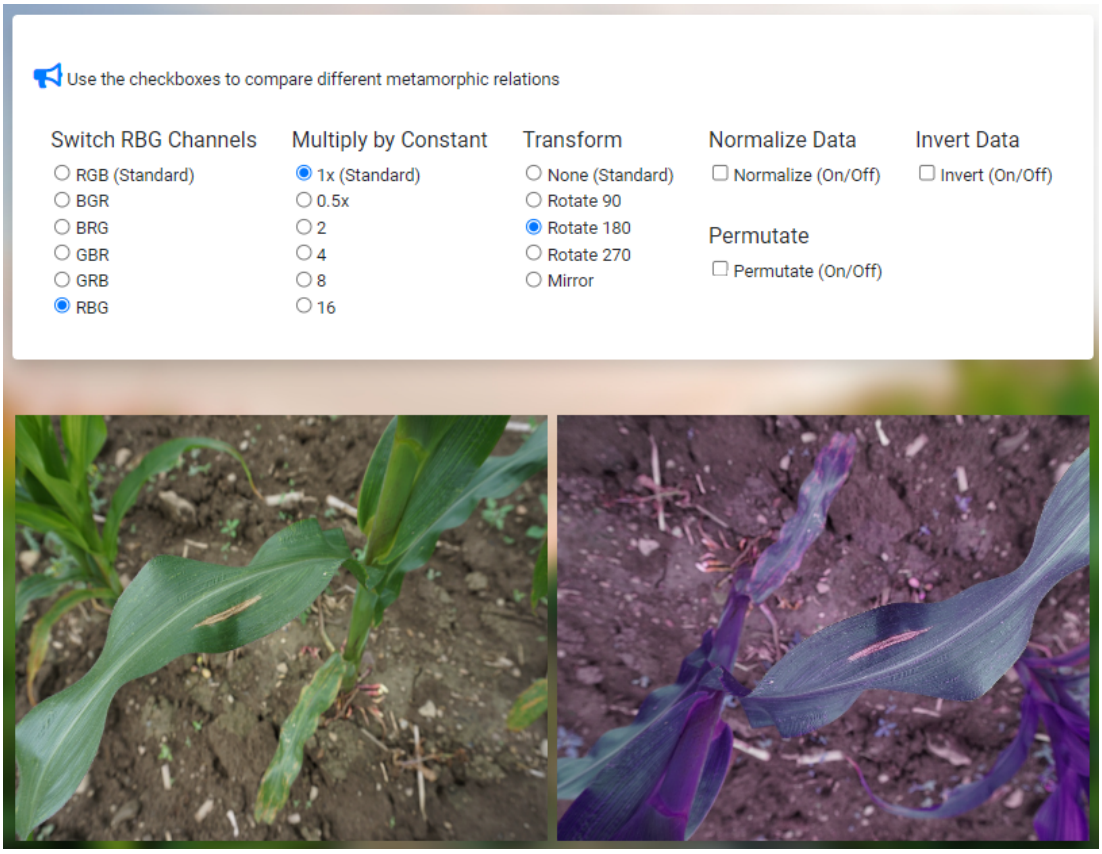

Figure 4.1: MR Selection

4.3 Differential Testing - Hyperparameter Selection

Another dimension of this ability is for the user to select their own hyperparameters. From menu.html, the second page available has options for the user to select values for kernel, C, and gamma (see Figure 4.2). The reason these three were chosen is because they are widely known to have the largest impact on model performance. This menu could be expanded with more hyperparameters at a later date. The user's selection is stored in session variables

and used to build both models.

This is considered a type of differential testing in which we keep track of all user hyperparameter selections and the results of the model, which can be compared against each other at a later time to evaluate the correctness of the model and the quality of input data. Each of the user selections, along with image classifications, MR selections, model confidence, and accuracy, are kept track of in ml_tracking.txt, which can be used to evaluate models that have been trained across all sessions in the application. This can be used in the future to determine the combination of hyperparameters and images used to train the best models.

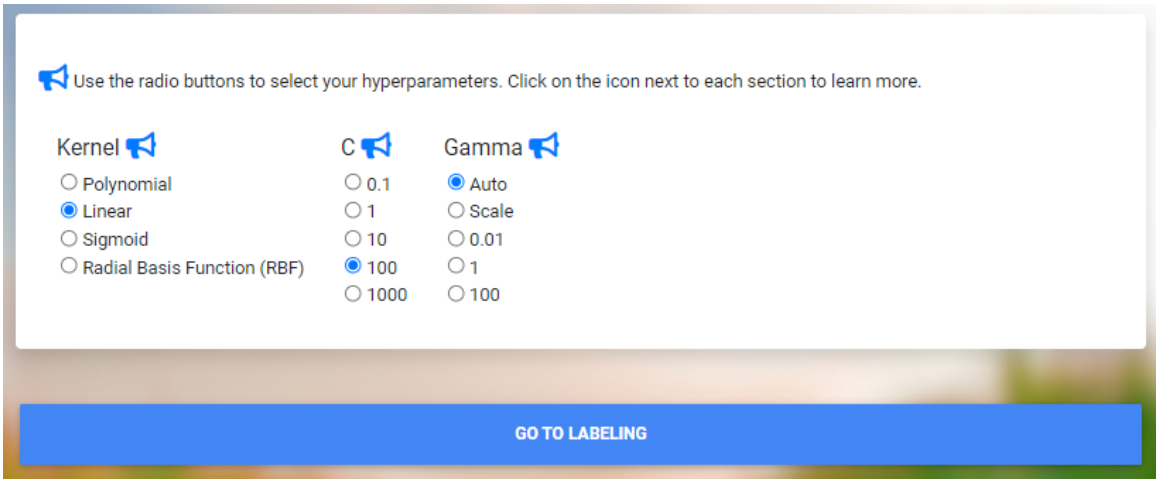

Figure 4.2: Hyperparameter Selection

4.4 Pretrained Model

To show the successful implementation of an SVM model for this data set and to compare the results of the user-generated model, we included the classification results from a pre-trained model. The pre-trained model is trained on about 1,600 images outside of the scope of the application and generates a much higher accuracy when compared to the user-trained models. The pre-trained model is stored in a pickle file that is easily accessible during the runtime of the application. The use of this pre-trained model led to the development of a new

Python class (ML_PreTrained - see Listing 4.1) which includes many of the same methods as the original ML class, the major exception being that no training takes place, while the other models are trained on the set of training images at each stage of the application (intermediate and final).

Listing 4.1: Pretrained Model Initialization

```
class ML_Pretrained:
    def __init__(self, train_data, preprocess):
        """
        Initialize the pretrained machine learning model.

        Attributes
        ----------
        preprocess : Python Function
            Function used to preprocess the data before model creation.
        X : pandas DataFrame
            The features in the train set.
        y : pandas Series
            The responce variable.
        ml_model : fitted machine learning classifier
            The machine learning model created offline on previous data.
        """

        # Get existing best trained model from stored pickle file
        pickle_filename = "pre_trained_models/best_trained_120_80.pkl"
        pickle_file = pickle.load(open(pickle_filename, 'rb'))
        self.ml_model = pickle_file
        self.preprocess = preprocess
        self.X = train_data.iloc[:,:-1].values
        self.y = train_data.iloc[:,-1].values
        self.X = self.preprocess.fit_transform(self.X)
```

A few critical issues that we ran into during deployment were matching the shape of the test data in the application and the size of the files in use. The shape of the training and test data used in the application must match the shape of the training data used for the pre-trained model. Also, the pickle files are too large to commit to GitHub, so these have been ignored using the .gitignore file. There are other methods that can be used to accomplish this task, for the sake of time and preparation for this thesis, we ignored the uploading of the pre-trained models.

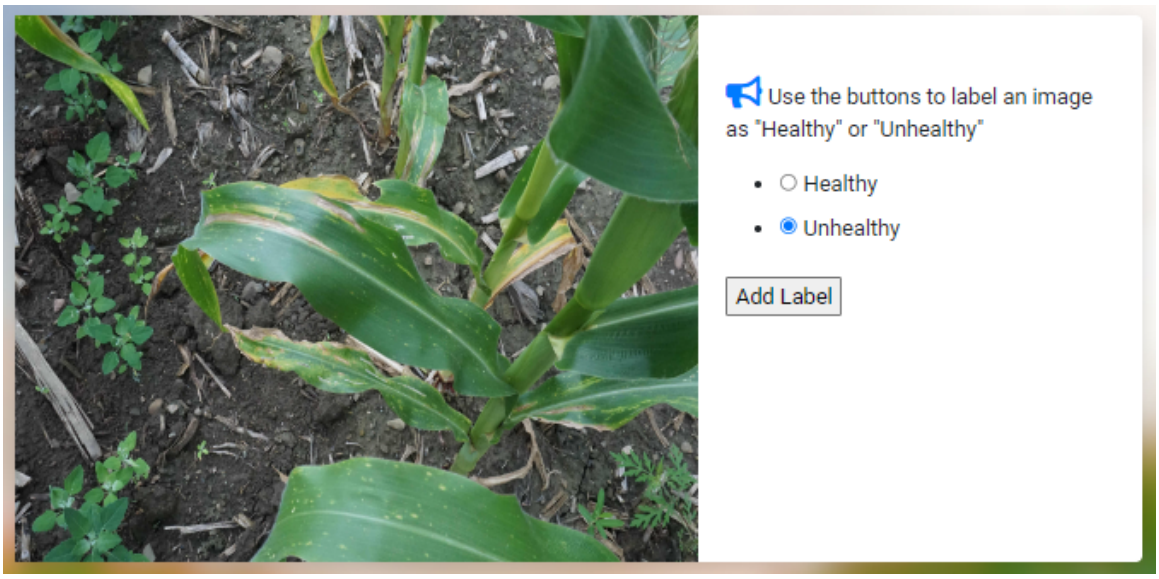

Figure 4.3: User Labeling

4.5 User Labeling - Final Results

The application allows the user to interact with each stage of the machine learning model creation. We mentioned the selection of hyperparameters earlier, but the user also will label the images that the model is trained on. Once the user has selected their metamorphic relations and hyperparameters, they are given ten random images to classify. The user selects a checkbox, labeling the image as either healthy or unhealthy. Based on the user selections, the model is trained using these labels and attempts to classify the remaining images in the data set.

Once ten images have been labeled, the original ML model is trained and confidence is generated. The confidence is a k-fold validation score which provides a metric for how confident the model is at classifying the images the user has labeled so far (this gives no indication of the remaining images). If the confidence is over 70% or at least 20 images have been selected, the user is sent to the final screen for the evaluation of all three models available in the application (see Figure 4.4). Otherwise, the user is sent to an intermediate

screen where they can view their current classifications, the confidence of the model, and the user's accuracy so far. The user can then continue to label the images in groups of five thereafter until one of the thresholds is met.

The overall flow of the application is not something that was provided in the scope of this work, another article covers this in more detail [18]. Many other improvements have been made to each stage of this application which are detailed in further sections.

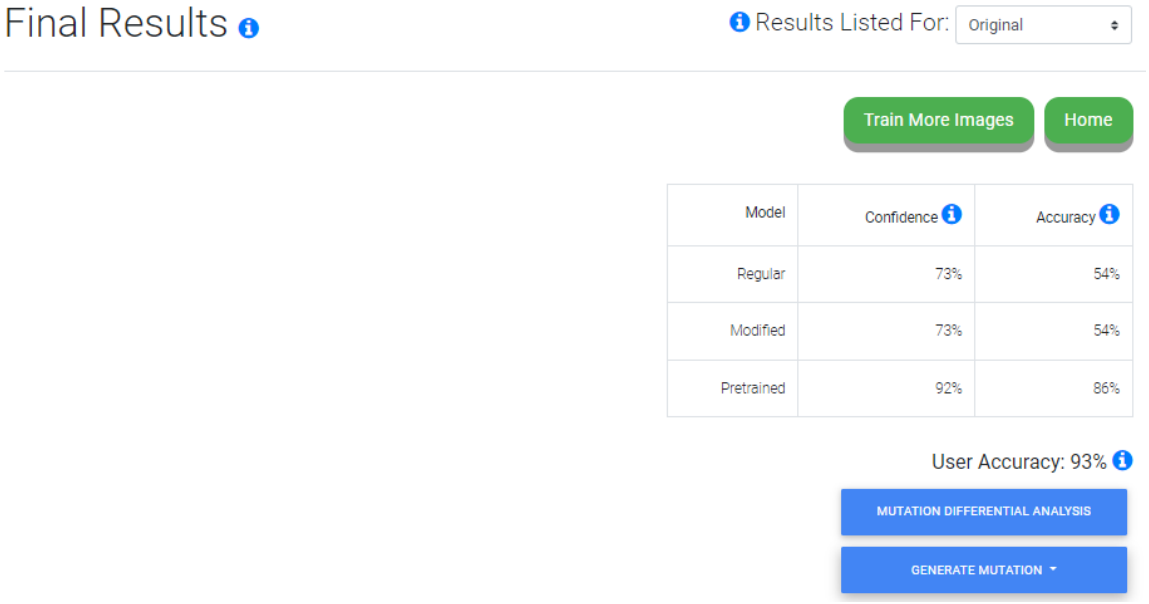

Figure 4.4: Final Results

4.6 Mutation Testing

Mutation testing is used widely to validate testing approaches. In mutation testing, bugs are introduced into an application to see if the available test data will find them. In this research, we started with using program mutations as a way to introduce bugs into a program and validate if metamorphic testing was able to catch these bugs. We realized later that mutation testing is better aligned with mutating the post-training ML model as this is at the heart of ML-based applications [19]. We switched to a mutation testing approach which we inject

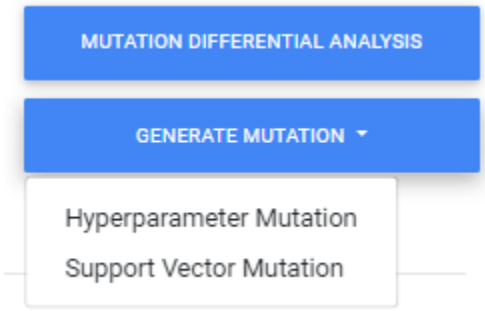

Figure 4.5: Mutation Options

model mutations so that the user can directly see the impact that these mutations have on the ML classification decisions.

Once the user reaches the final screen, they have the option to generate mutations for each machine learning model. The user can select to randomly mutate one of the hyperparameters or to randomly mutate one of the support vectors (see Figure 4.5). Each of the options may or may not have an impact on confidence, accuracy, and classifications for each model.

4.6.1 Hyperparameter Mutation via ChatGPT API

The hyperparameter mutation is done via a prompt given to a ChatGPT API. The AI model returns a JSON object which includes the values of each hyperparameter within the given machine learning model. The program iterates through the JSON object and updates all values according to the GPT response. Each value is compared to its previous version, with the differences being recorded for viewing later in the application.

Chat GPT is configured using the gpt_config.py file. The secret key and openai key can be found in the developer section of openai's API documentation and must be updated to enable use in the application [16]. All functions used related to the Chat GPT API are provided in the gpt_api.py file. The function generateChatResponse is simple to use which only requires the prompt that the user wishes to have answered and returns the GPT response (see Figure 4.6). A specific function to handle the hyperparameter mutation was

```python
def generateChatResponse(prompt):
    """
    Generates response from ChatGPT API call

    Parameters
    ----------
    prompt : String
        The prompt given to ChatGPT

    Returns
    -------
    answer : String
        Chat GPT response
    """

    messages = []
    messages.append({"role": "system", "content": "You are a helpful assistant."})

    question = {}
    question['role'] = 'user'
    question['content'] = prompt
    messages.append(question)

    response = openai.ChatCompletion.create(model='gpt-3.5-turbo',messages=messages)

    try:
        answer = response['choices'][0]['message']['content']
    except:
        answer = 'Error'

    print("GPT Answer: " + answer)

    return answer
```

Figure 4.6: Chat GPT Function

created, which takes as input one of the ML Models, and two Python lists (mutation list and class list). To generate a consistent response from Chat GPT, the prompt must be extremely thorough and specific in its instructions. Many different drafts were created before we saw consistent results. An example of the final draft is listed below for reference:

> You are working with the hyperparameters for an SVC() model trained via sklearn imported using "from sklearn import svm". Randomly change exactly one hyperparameter, do not remove any existing hyperparameters and leave probability=True. All hyperparameters should receive an equal chance of being selected for mutation. All previous prompts should have no impact on your decision. Only reply with an answer in the same form of the text given follow-

ing the colon and nothing more: {'C': 0.1, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0, 'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 0.0001, 'kernel': 'poly', 'max_iter': -1, 'probability': True, 'random_state': None, 'shrinking': True, 'tol': 0.001, 'verbose': False}

4.6.2 Support Vector Mutation

The support vector mutation is done using the Python library called random. This allows you to select a random integer or float within a given range. We first select a random index of the support vector, then we select a random real number within the range of 0.9 to 1.1. The support vector is then multiplied by this random real number. All metrics (confidence, accuracy, and new classifications) are re-calculated and stored for reference later.

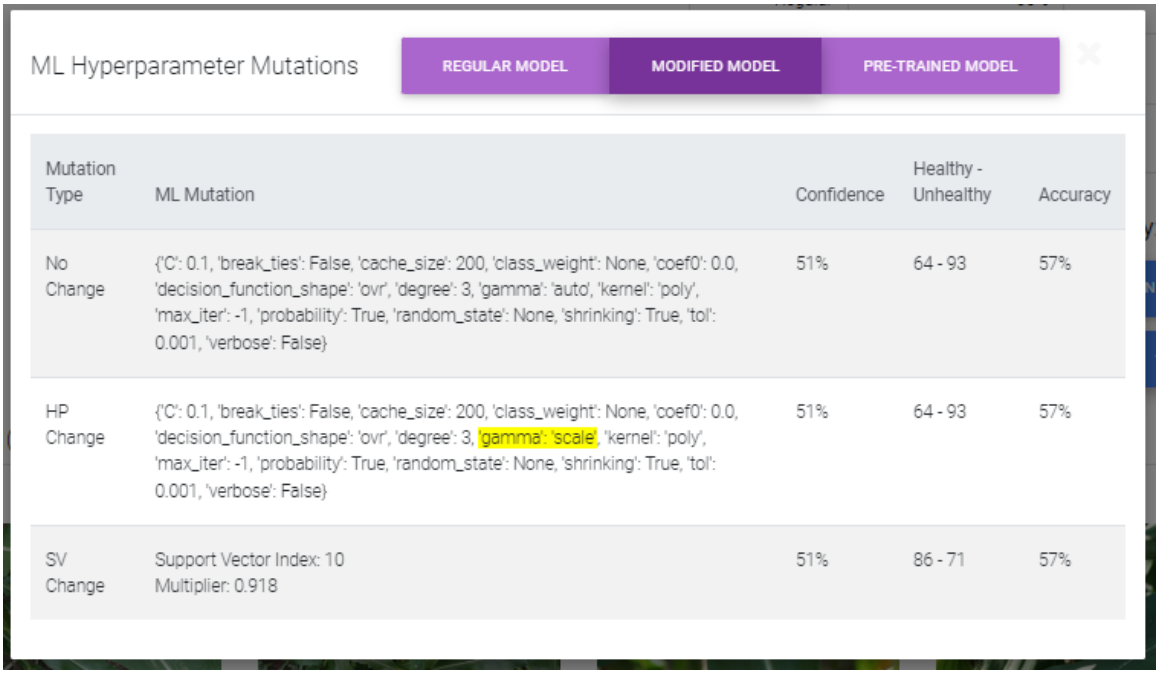

Figure 4.7: Mutation Differential Analysis

4.7 Differential Analysis Table

Metrics for confidence, accuracy, and future image classifications are gathered throughout each stage of the application. To review these metrics across each generation of the machine learning's life cycle, the user can click "Mutation Differential Analysis" from the final screen, which calls out the mutations across each generation as well as compares each of the metrics in a table that is easy for the user to visualize (see Figure 4.7). This table continues to grow indefinitely while the user continues to generate more mutations. The table resets once the user goes back to the home screen (resets the process entirely) or decided to train more images (can train images in increments of five until the data set is empty).

Interestingly, we have been able to find some support vector and hyperparameter mutations that did not impact the accuracy of classification of the test inputs. This underscores potential gaps in the test data that could have exposed such "bugs" in the model. Identifying additional test data needed to close such gaps is still a research problem [19].

4.8 Performance Improvements

Listing 4.2: Bitmap Image Scaling

```
# Library to open/resize images
from PIL import Image #to open images

# Get image, resize and reshape to flat array
img = Image.open("images_handheld_resized\\DSC00025.JPG")
img = img.resize((120, 80))
img = np.array(img).reshape(-1)
```

One of the major issues that we continued to face in this application was the balance of model performance and speed. The models tended to perform well and have much higher accuracy when the image sizes were larger, but this also slowed down the speed of the

application drastically. The first strategy we used was image condensing, which scaled the images down from about 960K pixels at full size to about 28K pixels for use in the application (see Listing 4.2). We found this is the lowest we could go without negatively impacting the results of the model.

The second strategy we used was declaring all data objects and machine learning models as global variables in the application. This allowed us to only collect the data once (at the beginning of the application) and not have to re-train the model at each stage of the application as well. The trade-off of memory for speed was ideal in this situation but would cause problems in scaling the application to multiple users down the road. This is a problem we leave open for future work if this application were to be deployed into production.

The last strategy we used to help speed up the application is limiting the number of images displayed on the final screen. Only ten test images are loaded for healthy and unhealthy, with the option to show more (10, 25, 50, or all - see Figure 4.8).

4.9 Usability Improvements

A few other improvements have to deal with the user interface. The labeling mechanism has been improved to reset the user's selection each time a new image is loaded on the screen. This will reduce bias as users are forced to make a selection each time instead of being allowed to select the same option as last time. Also, the Chat GPT API is available for use while the data is being collected for each image. This is the longest wait time the user has, so we've decided to give the user some tooltips related to metamorphic testing and SVM models (see Figure 4.9). The user can also prompt GPT to provide more insight, and additional examples, or to write their own question during the loading time.

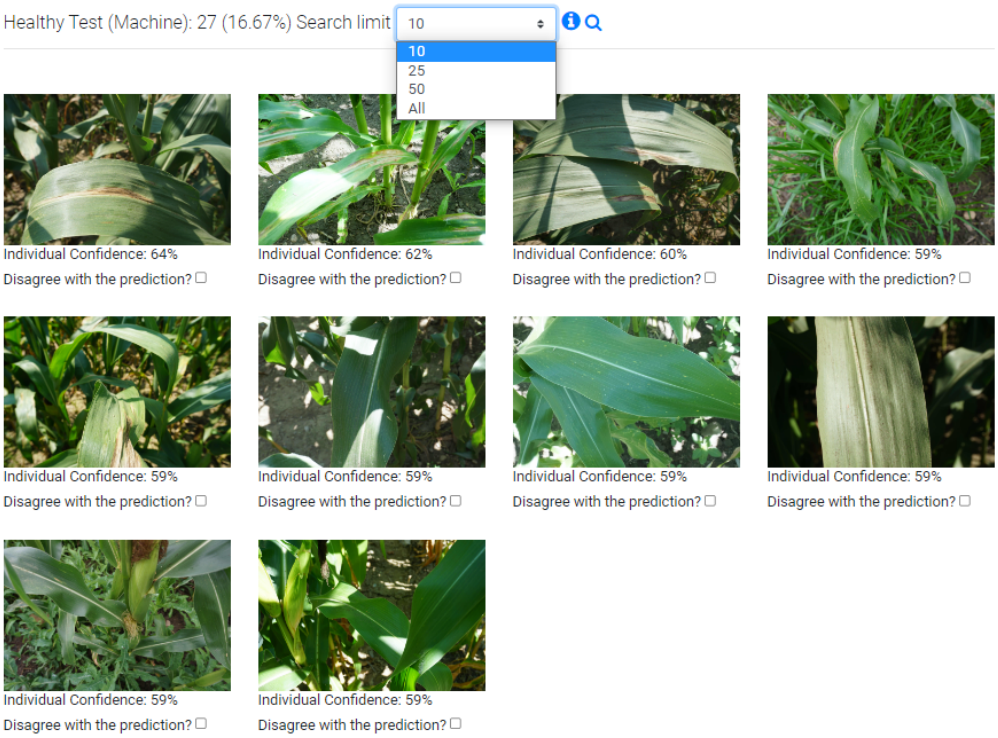

Figure 4.8: Scaling Image Loading

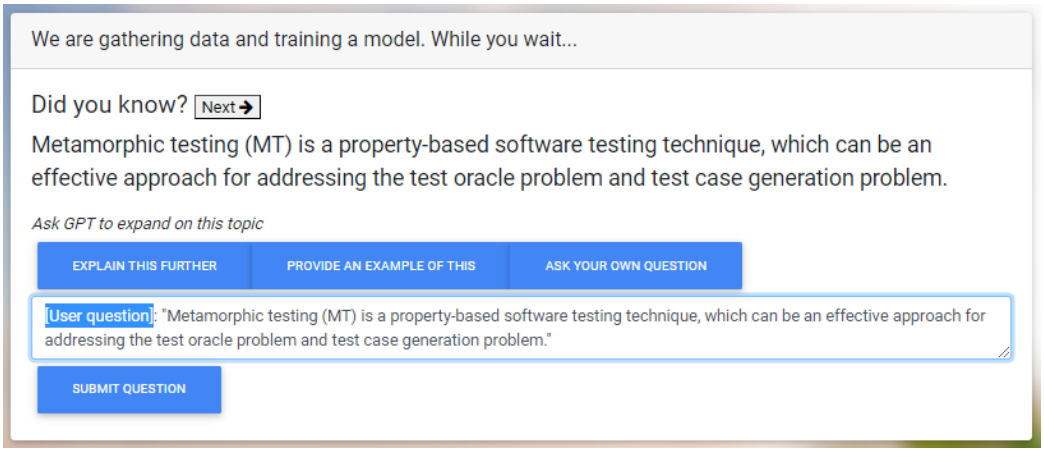

Figure 4.9: Tooltip Enhanced with Chat GPT

Chapter 5
Conclusion

5.1 Limitations

There are several limitations to our research that should be acknowledged. First, the scope of the project is limited to a specific leaf data set, which may inhibit our research from being generalized. Additionally, the interactivity of the application presented challenges in terms of model accuracy. While larger bitmaps yield more accurate results, they also require a longer training time, which hindered the speed of the application. Furthermore, our study only tested the methodology against SVM, which may not provide a comprehensive evaluation of its effectiveness for all ML algorithms.

5.2 Summary

In summary, our research project has made an impact by being the first application to evaluate Metamorphic Relations on this specific set of images. We assessed the feasibility of using pixel inversion as a Metamorphic Relation and conducted Machine Learning mutations post-training using a Chat GPT API, the first to do so. This approach provides a framework for evaluating the robustness of an ML model under test and allows for comparisons with new hyperparameters after the training process. Our project's primary focus was to explore the use of metamorphic testing, mutation testing, and differential testing to enhance Machine Learning models. We successfully developed an application that demonstrates the potential uses of these testing techniques in practice. Moreover, we provided a no-code solution for training an image classification model using this data set, which can serve as a framework

for other image sets. This project showcases the potential for further advancements in Machine Learning model testing and improvement.

5.3 Future Work

There are many areas of this work that can be expanded on in the future. First, a continued study of effective ways of conducting mutation testing is needed. While we are able to identify situations where some support vector or hyperparameter mutation had no effect of the classification, we have not been able to leverage this information to guide us in better understanding the model, e.g., where the classification boundaries are. Additional investigation into the mathematics of support vectors are needed to fill the gaps in testing data.

Second, we propose extending this platform to include other machine learning algorithms like RandomForest, K-Nearest Neighbor, etc., with adjustments to the set of hyperparameters. These can be explored in the future to better understand the robustness of our method. By using bitmap inputs to the model, it can also potentially be extended to deep learning algorithms such as Convolutional Neural Networks (CNN). A key problem that need to be addressed is that more sophisticated neural networks tend to have hundreds or even thousands of hyperparameters. An approach for determining the values for collections of hyperparameters will need to be developed.

Third, the incorporation of deep learning models opens up the possibility for additional algorithms such as Siamese networks [5]. Siamese networks contain multiple identical subnetworks and are trained with a relatively small training set to determine if two images are the same rather than performing a classification. We propose to study what ML testing techniques will work well for similarity checking rather than classification. This will help in addressing real-world challenges where only a limited number of samples are available for each class. By pursuing these future research directions, we hope to contribute to the

continued development and refinement of ML testing and improvement techniques, as well as to extend the practical applications of our approach.

Bibliography

[1] A. Borchers. Agro-AI testbed GitHub repository. https://github.com/PW-AlexBorchers22/agro-masters-project.

[2] J. Brownlee. Save and load machine learning models in Python with scikit-learn. https://machinelearningmastery.com/save-load-machine-learning-models-python-scikit-learn/, 2016.

[3] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, 1980.

[4] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology, 1998.

[5] D. Chicco. Siamese neural networks: An overview. *Artificial neural networks*, pages 73–94, 2021.

[6] J. A. Clark. Pillow (pil fork) 9.5.0 documentation. https://pillow.readthedocs.io/en/stable/, 2023.

[7] J. Dowling. Guide to file formats for machine learning: Columnar, training, inferencing, and the feature store. https://towardsdatascience.com/guide-to-file-formats-for-machine-learning-columnar-training-inferencing-and
2019.

[8] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 574–586, 2018.

[9] S. Dutta, A. Shi, and S. Misailovic. FLEX: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 603–614, 2021.

[10] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder. 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*, pages 118–128, 2018.

[11] M. A. Gulzar, Y. Zhu, and X. Han. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 71–80. IEEE, 2019.

[12] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–498. IEEE, 2020.

[13] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161. IEEE, 2019.

[14] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010.

[15] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[16] OpenAI. Chat GPT API documentation. https://platform.openai.com/docs/api-reference, 2023.

[17] D. Orn, L. Duan, Y. Liang, and S. Kratochvil. Agro-AI capstone GitHub repository. https://github.com/skratochvil/Ag-AI, 2020.

[18] D. Orn, L. Duan, Y. Liang, H. Siy, and M. Subramaniam. Agro-ai education: artificial intelligence for future farmers. In *Proceedings of the 21st Annual Conference On Information Technology Education*, pages 54–57, 2020.

[19] A. Panichella and C. C. Liem. What are we really testing in mutation testing for machine learning? a critical reflection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 66–70. IEEE, 2021.

[20] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.

[21] Scikit-Learn. GridSearchCV API documentation. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[22] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie. Multiple-implementation testing of supervised learning software. In *Workshops at the thirty-second AAAI conference on artificial intelligence*, 2018.

[23] T.-W. Weng, H. Zhang, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, and L. Daniel. Evaluating the robustness of neural networks: An extreme value theory approach. *arXiv preprint arXiv:1801.10578*, 2018.

[24] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[25] T. Wiesner-Hanks, E. L. Stewart, N. Kaczmar, C. DeChant, H. Wu, R. J. Nelson, H. Lipson, and M. A. Gore. Image set for deep learning: field images of maize annotated with disease symptoms. *BMC research notes*, 11(1):1–3, 2018.

[26] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 826–837, 2020.