Contents

1. Introduction:

Data storage is widely used and in high demand in today's interconnected business ecosystem. For example, each day Walmart stores potentially several petabytes of data due to the millions of transaction records, updates of inventory stock, information of new customers, etc. Merchants are always required to choose the best storage solution to avoid unnecessary overhead costs and provide more comfort services for customers. Their solution is storage management systems using cloud, tape, and disk.

Currently, cloud storage seems to be the best option for many companies. It is affordable compared to traditional disk storage since it doesn't need to be operated by the business itself and pays for itself in terms of ownership, maintenance, and operation of servers. Merchants can rent these servers provided by companies like Amazon and Microsoft. Most importantly, customers can quickly access data stored on the cloud at any time, anywhere, with many different types of devices.

Disk and tape storage are the other forms of storage systems IBM Spectrum protect uses. Disk is one of the better storage solutions for quickly accessing data. It is quite expensive compared to cloud or tape but if a customer wants to access their data immediately then disk would be the right option. Tape ,on the other hand, is a cheaper form of storage however it takes longer to access by the customer, therefore, it is mainly for data that doesn’t need to be accessed immediately.

Our project is sponsored by Daniel Boros who is a staff software engineer in the IBM Spectrum Protect project. IBM Spectrum Protect is a generalized monolithic server with cloud capabilities, which is designed to simplify protection for large amounts of data hosted in physical, virtual, software-defined, and cloud environments for all customers. Also, Spectrum Protect simplifies backup administration, improves efficiencies in the backup process, and enable scalability to an entire enterprise of inputs. Our project is committed to improving secure enterprise level storage for IBM by providing a good solution for classifying data into the appropriate storage tier.

As of now, the data that's being stored by Spectrum Protect is often miscategorized. An example would be data that should be stored in tape is now stored in disk. This means that after a period of time it will get demoted to a lower tier by policies in place to help categorize this data. These policies take some time to take action and during the time that this data is getting demoted, overhead costs for IBM increase the longer the data is in an incorrect tier. If the opposite happens and data that should be on disk is now on tape, customers might not be able to access their data quickly enough and depending on the situation could waste time and money for them as well. The costs for Spectrum Protect increase as more data is stored incorrectly which is why handling the storage tier on ingestion is crucial. Our project is important because it can save time, money, and effort that can be handled by this pipeline.

So far we have the desired pipeline that can preemptively categorize data uploaded to Spectrum Protect into its correct storage tier to avoid unnecessary overhead costs. To ensure our product is reliable and can categorize most data into its correct storage tier, we need to perform multiple software testings on the pipeline and make sure it satisfies all the goals and requirements from our sponsor and our software design document. If a file that should be stored in tape is now miscategorized into the disk. After a period of time it will get demoted to a lower tier by policies in place to help categorize this data. These policies take some time to take action and during the time that this data is getting demoted, overhead costs for IBM increase the longer the data is in an incorrect tier. If the opposite happens and data that should be on disk is now on tape, customers might not be able to access their data quickly enough and depending on the situation could waste time and money for them as well. The costs for Spectrum Protect increase as more data is stored incorrectly which is why handling the storage tier on ingestion is crucial. So implementing multiple software testings is important for our product because it can make sure our product does function correctly under any given inputs without any unexpected crashes or bugs. Also, it is really important process to roll out our product to its final stage.

In our software testing plan, we will mainly focus on three parts: unit testing, integration testing and usability testing.

2. Unit Testing:

Unit testing is the smallest testable “unit” of a software. The purpose is to validate the behavior on a micro-level, that is, to validate the unit works as expected only looking at the behavior of the unit. For example, a class may have many methods and you would use unit tests to validate how each individual method behaves not how the class overall behaves. The result, if every unit is tested properly, is assurance each element is behaving as expected and makes finding future bugs easier to identify.

For our project, we will be using pytest. Pytest is a framework for testing relying on assertions in python. An assertion is a statement that must hold true and therefore is the tested part of the tests. We are using pytest because of its ease of use when programming, ease of integration, and the large level of documentation because of the large following who use it. We will focus on test coverage as a metric to ensure everything is tested. While test coverage does not cause a system to be well tested it does correlate with a well-tested system. We do acknowledge each unit needs to have proper domain and range identified and elements that fall in and out of those need to be tested for proper behavior; otherwise, test coverage has no real value.

The units we will be testing can be identified by breaking down two of our three main modules. In breaking down our system into units we end up with the following table (figure 3.1). Notably, one type of function is missing that is present in our Learn module. These are model function wrappers that pass to a profiling API we are using. We are not testing these functions because this would serve to only test the API we are using. We are not unit testing the Driver aspect of our project because it only serves to combine the other modules into a pipeline. It is more appropriately tested in integration testing. The Display module will not be described here because it serves only to draw the GUI elements of our project. This is more appropriately tested in usability testing

Module	Learn	Process
	linearSVC	combine
	multinomialRegression	extract_features
	randomForest	
	gradientBoosting	

	save_model	
	load_model	
	validate_model	
	data_metric	

Figure 3.1

To test each unit properly we have identified equivalence partitions and boundary points for each. We will outline them in order of the modules above; Learn, Process, Driver, Display.

The learn module consists of two types of functions, model creation functions and slots to buttons that are part of the GUI in Display. We will start with the model creation functions. All of these functions take many parameters that are used in creating the model. Many of these parameters repeat in multiple places and represent the same thing in each of these places. For the readability of this document, all of the parameters and their equivalence partitions and boundary points will be defined once in table (Figure 3.2) and referenced as necessary.

The behavior of our model creation functions are expected to behave with near equivalency. Because of this, we will not define the test for each but the template of the test that will be implemented for each. The functions included in this are linearSVC, multinomialRegression, randomForest, and gradientBoosting. The function signatures are:

- def linearSVC(df, feature_list=['BFSIZE', 'HDRSIZE', 'NODETYPE'], maxIter=100, regParam=0.0, threshold=0.0,)
- def multinomialRegression(df, feature_list=['BFSIZE', 'HDRSIZE', 'NODETYPE'], maxIter = 100, regParam = 0.0, elasticNetParam = 0.0, threshold = 0.5)
- def randomForest(df, feature_list=['BFSIZE', 'HDRSIZE', 'NODETYPE'], maxDepth = 5, numTrees = 20, maxBins=32, impurity='gini', subsamplingRate=1.0)
- def gradientBoosting(df, feature_list=['BFSIZE', 'HDRSIZE', 'NODETYPE'], maxIter=20, stepSize=0.1, maxDepth=5)

Parameter	**Type**	**Equivalence Partitions**	**Equivalence Partition Examples**	**Boundary Values**
df	Spark DataFrame	All valid Spark DataFrame of doubles with a label column All valid Spark DataFrames of doubles All valid Spark DataFrames containing at least one non-double column All invalid Spark DataFrames	A valid Spark DataFrame of doubles with a label column A valid Spark DataFrame of double missing a label column A valid Spark DataFrames with a column of strings A Spark DataFrame formed from malformed CSV	A valid Spark DataFrame of doubles with a label column A valid Spark DataFrame of double missing a label column A valid Spark DataFrames with one column of non-doubles A Spark DataFrame formed from malformed CSV
feature_list	List of Type String	All lists of strings that are columns in df All lists of strings that have entries which are not columns in df All lists with at least one non-string entry	When columns in df are: (this, is, one) [‘this’ ‘is’ ‘one’] [‘this’ ‘is’ ‘not’] [‘this’ ‘is’ 0]	A list of strings that have no entries which are not columns in df A list of strings that have an entry not in columns in df A list where one entry is non-string
maxIter, maxDepth, and numTrees	Integer	Positive Integers Non-Positive	12 -12	1, memory limit 0, - memory

		Integers		limit
elasticNetParam, regParam, setpSize, subsamplingRate, and threshold	Double	Non-negative real number Negative real number Non-real number	12.7 -12.7 12j	0 -0.1 x 10^-n real
impurity	String	‘gini’ || ‘entropy’ not ‘gini’ || ‘entropy’	‘gini’ ‘seven’	‘gini’ || ‘entropy’ not ‘gini’ || ‘entropy’

Figure 3.2

For each model function, we will test each case as defined above for each. The expected output will be the same for each test. A tuple which contains the results of testing the model, the model, and the data used to validate the model. The output will be tested that all expected types are present, the results have all expected metrics present, and that the results are well formed for transformation into a log file. The model and data will only be tested for the correct type as testing any further will test be testing the Spark API.

The other functions can be more easily individually defined but there is still overlap in the parameters. The parameters can be defined by the following table (Figure 3.3).

Parameter	**Type**	**Equivalence Partitions**	**Equivalence Partition Examples**	**Boundary Values**
model	Spark ML Model	A Spark ML Model Not a Spark ML Model	RandomForestModel ‘Apple’	Is a Spark ML Model Is not a Spark ML Model

metrics	Dictionary	A dictionary object Not a dictionary object	{‘is’: 6} ‘not’	Is a dictionary object Is not a dictionary object
file_location	String	A string pointing to a valid file location A string pointing to an invalid location String not pointing to a location	/anyone/can/write /sudo/only/ apple	A file format with access A file format without access Any other string

Figure 3.3

The save_model function has the following function signature def save_model(model, metrics, file_location). We can test the model, metrics and file_location parameters as described and to check for proper output we can check there was a file saved in the specified location. The load_model function has the following function signature load_model(file_location). We can test the inputs for file_location as defined above and for correct output, we can check the returns of the function the model, and metrics. We can check that the return tuple has the correct types. Any further checking will be testing the API used to load and save the model. The validate_model function has the following function signature validate_model(model, metrics, testData) the first two parameters can be seen in Figure 3.3, but testData can be defined as df in Figure 3.2. The return of the function is an altered version of metrics holding the results from the validation of the model. To test for correct output we can test three things 1) that the returned type is a dictionary object 2)that it has all elements expected from validation and 3) that elements relating to training the model are equivalent in both the returned dictionary and in the metrics parameter. The last function is data_metric which has the function signature data_metrics(metrics, trainingData, testData). The parameters trainingData and testData can be defined as df in Figure 3.2. The outputs of the function is an altered metrics object and can be

tested by ensuring 1) the returned object is of type dictionary 2) the dictionary has the altered elements as expected 3) the dictionary has expected elements unchanged compared to the metrics parameter.

The Process module has two functions combine and extract_features. Combine serves as a helper for extract_features. Both functions have only feature_list as a parameter to have inputs tested. This is the equivalent of feature_list in Figure 3.2 with the caveat that df, in this case, is a DataFrame combined in the combine function instead of being a parameter. Combine returns a Spark Dataframe. We can test for correctness by testing two things 1) The output is of correct type 2) The columns of the returned DataFrame contain the elements of feature_list. Extract_features also returns a DataFrame but it can be more thoroughly tested as there are more requirements. The following can be tested 1) the return is of correct type 2) the columns contain only the elements of feature_list as well as a column ‘label’ 3) there are no null values in any column 4) all columns are of type float.

3. Integration Testing:

Integration testing can be considered as a level of testing above unit testing. It is testing the combination of units and testing whether this combination is behaving as expected. The importance is to expose defects or faults in the interactions between components. The idea being all individual components may work as expected but the components are combined in such a way that it fails to function properly. The way we are approaching our integration testing will be with a bottom-up approach. The strategy will be to build up our end integrated system by testing each step of the build up. We will use pytest for our integration tests, the scope of the tests will just cover more components.

There are three main steps that happen in our system pipeline. The preprocessing of data, creation and validation of a model, and the display of the metrics. There is limited integration in our system so we only have two steps of integration that we can test. The two components for testing will be the combination of 1) the preprocessing of data and creation/validation of a model 2) combination 1 and the display of the metrics.

To properly test our first integration we will take a subset of our code in our Driver module and test the output of the model creation is as expected. This code will have our extract_features function and the wrapper functions for our models. We will test that 1) we did return a model and metrics as expected 2) the metrics have all data we expect including:

- Model Parameters
- Learning Curve
- Model Name
- Features
- Accuracy Metrics
- Data Distribution

To test the next step of our integration we need to test the integration of our whole system. This will be the processing of data, the model creation, and the display of the model information. We can do this by directly testing the Driver module main function. We would need to test the GUI is running as expected by using the pytest-qt plugin. With the plugin, we can test all functionality of the GUI by simulating user interaction as well as test that expected elements in our GUI are present.

4. Usability Testing:

Usability testing is a necessary process for any large scale project in software engineering particularly when the software is being provided to an external client. It is important to ensure that once an end product has been handed over that the user can actually use it in the way it was intended to be used with ease. To ensure that the software we provide to our sponsor is usable, we will provide a sample group from to test our product. To be able to gather results that will reflect our end users, we must be careful about the population from which we will choose the sample from.

Daniel Boros, our sponsor, is currently our sole envisioned user. For good measure we should assume that he will not be the only person to ever work on this project therefore we should assume our end user to be any staff software engineer at IBM in Tucson. Unfortunately a team of software engineers is not at our disposal to test the usability of our software. Instead we must choose a population that is more readily available. Initially we thought to choose a sample from a population of computer science students at NAU. However, not every computer science student has the equivalent knowledge of a software engineer

working for IBM therefore we had to slim down the population. We decided it would be best to choose a sample from a population of computer science students who are either taking their capstone currently or have completed all the requirements to take their capstone and would theoretically start at the beginning of next semester. The sole qualification for participants would be knowledge of linux distributions, specifically adeptness with terminal operations. We determined that this was the best population to choose as we would have a group of at least 30 qualified students from which to choose. Based on the amount of feedback we expect to receive from each participant, we determined that a study group of approximately four to six participants would be appropriate for this study.

During the testing phase, there will be a set of guidelines in which we would have to follow in order to get the best results and feedback. To do this properly we must have a certain set of instructions given to test subjects as to not give them step by step instructions on how to do everything. In our case the first thing to do would be to ask the subject to download our project from our github website. This should be a simple task easily completed by all of our participants. Once the participant has a local copy of the project, we should ask them execute the docker container where all operations can be run. Once testing of the docker is complete, we will provide a pre-existing environment where the participant will be asked to complete a set of tasks:

- The participant will be told to run the driver file with random forests model with desired features that will vary per participant with parameters maxDepth set to 5 and numTrees set to 20. Participant will also be told they can use --help to look at our documentation
- Participant will be told to export data to the log file
- Participant will be told to run the GUI and have a visual display of all the metrics and graphs generated by the learning and testing process
- Once in the GUI, participant will be asked to save a model, compare two models, and re-test a model

As to avoid self report bias, a team member will serve as an evaluator for the participant and lead the usability testing as well as record the time to complete each section. Some questions we might have for the participant after the test would be:

- Did you feel as though creating and training the model was straight forward and a somewhat easy task to complete?

- If there was one thing you could change to improve usability during this step what would it be?
- How useful was our documentation?
- Was exporting the data and displaying the GUI straight forward and a somewhat easy task to complete?
- If there was one thing you could change to improve usability during this step what would it be?
- How difficult would you say it was to navigate through the GUI's features? If any section was difficult, what was wrong with it and what would you do to improve it?

The decision based on the feedback of the participant should be discussed among the team and dealt with accordingly. Once the feedback of the participant has been taken into account and has been determined that a change is required, the change will be implemented. If a participant does not give any constructive feedback but the time taken to complete the tasks was unnecessarily long then follow up questions might be necessary to pinpoint exactly where the delay occurred.

We hope to gather all participants together to study the testing phase at the same time if possible. Our testing process involves subjects who have a background in computer science and a higher than average problem solving ability therefore our directions and questions can be a lot more vague than most. Being able to discuss software specific details with participants will greatly increase our ability to improve our product's overall usability.

We hope that this testing phase can provide us with helpful feedback and suggestions for change so that we can deliver a better, easy to use product to our sponsor.

3. Conclusion:

In conclusion, we created this software testing plan for the purpose of performing multiple software tests for different components of our pipeline and verifying that all the components of our pipeline can communicate correctly and run as it was intended to. More specifically, we use three methods of software testing mentioned above to look into our code and fix minor bugs. In the unit testing, we can write specific tests for each function of our pipeline and make sure these “units” work correctly. Then we want to use the integration testing to check that all the modules perfectly worked together. At last, we will perform usability testing to ensure that the usability of our end product is suitable for our sponsor.