Testing For Trust User Guide

****The following steps are tried and tested on Windows10 OS****

A. Installation of "tft" (testing for trust) python library from **.whl** (wheel) file:

Step1: If anaconda is not installed, then download the latest version of Anaconda (windows installer) and then make a Python 3.6 environment

Step2: Extract the dist_cpu.zip (if your machine has only CPU) or dist_gpu.zip (if your machine has GPU) from the git repository. This will extract a folder that will have **.whl** file.

Note: On GPU machines, this tool has been found working correctly with **Tensorflow 1.10.0**, **cuda 9.0 and cudnn7.** If cuda and cudnn are not installed, they need to be installed.

Step3: Activate the new python 3.6 environment from windows command prompt or anaconda command prompt.

Step4: Execute the command `pip install <path to .whl file>` to install

Step5: To verify successful installation, open python interpreter and execute `import tft` to see if the library gets imported successfully.

```
Python 3.6.7 (default, Dec  6 2019, 07:03:06) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import tft
>>>
```

B. Apache tomcat setup for visualizing TFT results

Step1: Install apache tomcat 9 using Windows Installer

Step2: Extract the teachntest.zip file. This will extract a folder named "**teachntest**"

Step3: Copy this folder and paste under tomcat's **webapps** folder

C. How to use this library and visualize results?

1. A user (tester/developer/data-scientist) who would like to use this library first need to create a python file and import necessary modules (that abstracts methods against which model will be tested). For example, see the screenshot below:

```
# User imports our library
from tft.robustness_image import AdversarialInputs
from tft.generalization_image import FourierRadialFiltering, ObjectRecognitionWeakSignals, RotationTranslation
from tft.interpretability_image import ModelInterpretation
```

2. In the same file, the user must define a class that abstracts the model to be tested.

This model class must have following attributes.

a. Input image height as per the model
b. Input image width as per the model
c. Input image channels as per the model

d. A TensorFlow placeholder for input images
e. A TensorFlow session object (with model loaded)
f. The logit tensor of the model

An example below for InceptionV3 trained using TensorFlow's slim API:

```
# The Model class must have following attributes
self.image_size_height = 299  # Height of the image as per the model
self.image_size_width = 299  # Width of the image as per the model
self.num_channels = 3  # Number of channels in the image

# sess: a tensorflow session with the model (graph plus weights) loaded by a user
# logits: the model's output tensor i.e. the output of the softmax
self.sess, self.logits = self.load_model(modelpath, model_file)
# An input image placeholder
self.x = self.sess.graph.get_tensor_by_name("x:0")
```

This model class must have following functions.

__init__ (self, modelpath, model_file)

This __init__() method of the model class instantiates the model being audited with the above parameters.

Args:

modelpath: the absolute path to the model excluding file name.

model_file: the file name corresponding to the model (Example: <filename>.meta)

For Example, if we audit InceptionV3 model, the __init__() function would look like below:

```
class Model:
    """
    Define a Model to be loaded by the user.
    param logits: the model's output tensor i.e. the output of the softmax
    param sess: a tensorflow session with the checkpoint/model loaded by a user
    """

    def __init__(self, modelpath, model_file):
        # The Model class must have following attributes
        self.image_size_height = 299  # Height of the image as per the model
        self.image_size_width = 299  # Width of the image as per the model
        self.num_channels = 3  # Number of channels in the image

        # sess: a tensorflow session with the model (graph plus weights) loaded by a user
        # logits: the model's output tensor i.e. the output of the softmax
        self.sess, self.logits = self.load_model(modelpath, model_file)
        # An input image placeholder
        self.x = self.sess.graph.get_tensor_by_name("x:0")
```

pre_process (self, imgs)

This method should have an implementation for any pre-processing mechanisms for images for the model to predict appropriately.

For example, if we audit InceptionV3 model, the pre_process () function would look like below if we are using image pre-processing utilities from Keras for InceptionV3.

```
@staticmethod
def pre_process(imgs):
    """
    # The Model class must have a pre-process method
    :param imgs: A batch of images of shape (nSamples, h, w, c)
    :return: A batch of pre-processed images of shape (nSamples, h, w, c)
    """
    images_pre_processed = []
    for a_image in imgs:
        # a_image is of shape (dim1, dim2, channels)
        a_image = np.expand_dims(a_image, axis=0)  # line reshapes to (1, dim1, dim2, channels)
        # Pre process the input image as per inception_v3
        images_pre_processed.append(inc_net.preprocess_input(a_image))
    return np.vstack(images_pre_processed)  # Stack the images one below the other (rows)
```

Args:

imgs: A batch of images in the form of a list of NumPy arrays. Where each NumPy array represents an image of shape (height, width, channels).

This function should return:

Numpy ndarray of shape (nSamples, height, width, channels). i.e. A stacked batch of images.

rev_preprocess (self, **images**)

This function should have an implementation to convert back pre-processed images to its original form.

For example, if we audit InceptionV3 model, the rev_preprocess () function would look like below:

```
def rev_preprocess(self, im):
    """
    The model class must have reverse preprocess rev_preprocess() method

    :param im: A single image or a batch of images of respective shapes (h, w, c) or (nSamples, h, w, c)
    :return: A single image or a batch of images of respective shapes (h, w, c) or (nSamples, h, w, c)
    """
    im += 1
    im *= 128
    return im
```

predict (self, **images**)

This function should have an implementation to predict the outcomes for a batch of images. This function will in turn call pre_process function for processing inputs before passing it for actual prediction logic.

For example, if we audit InceptionV3 model, the predict () function would look like below:

```
def predict(self, images):  # images will have values(pixel values) from 0 - 255
    """
    The Model class must have a predict method
    :param images: a list of image tensors of shape (nSamples, H, W, C) ; where H represents height, W represents
           width and C represents channels of an image respectively

    :return: array of arrays of predictions for each image sample.
          i.e. [[p(class0/image0),p(class1/image0),....,p(classN/image0)],
          [p(class0/image1),p(class1/image1),....,p(classN/image1)],.......,
          [p(class0/imageN),p(class1/imageN),....,p(classN/imageN)]]
    """
    probabilities = tf.nn.softmax(self.logits)
    print(images.shape)
    pre_processed_images = self.pre_process(images)
    print(pre_processed_images.shape)

    return self.sess.run(probabilities, feed_dict={self.x: pre_processed_images})
```

Args:

images: a list of image tensors of shape (nSamples, Height, Width, Channels)

This function should return:

Array of arrays of probabilities for each image sample.

i.e.

[[p(class0/image0), p(class1/image0),...., p(classN/image0)],
[p(class0/image1),p(class1/image1),....,p(classN/image1)],.......,
[p(class0/imageN),p(class1/imageN),....,p(classN/imageN)]]

Note: p(class0/image0) ➔ probability of image0 being class0

p(class1/image0) → probability of image0 being class1 and so on.

load_model (self, path_to_model, model_file_name)

This function should have an implementation to load the model into a TensorFlow session.

For example, auditing an InceptionV3 model would have this function as below.

```
def load_model(self, path_to_model, model_file_name):

    sess = tf.Session()
    saver = tf.train.import_meta_graph(os.path.join(path_to_model, model_file_name))
    LOGITS_TENSOR_NAME = 'InceptionV3/Logits/SpatialSqueeze:0'
    saver.restore(sess, tf.train.latest_checkpoint(path_to_model))
    logits = tf.get_default_graph().get_tensor_by_name(LOGITS_TENSOR_NAME)
    print("Checkpoint loaded..")
    print("logits..."+str(logits))
    return sess, logits
```

Args:

path_to_model: the absolute path to the model excluding file name

model_file_name: the file name corresponding to the model (Example: <filename>.meta)

Note:

Sometimes, the number of arguments of the __init__() & load_model () methods may slightly differ based on the model we are testing and the way to load it. Please refer to the respective python files from the below links that was prepared while testing Resnet50 & DenseNet121 architectures as reference:

Resnet50:

https://github.com/testingForTrust/tft/blob/master/userModel_resnet50.py

DenseNet121:

https://github.com/testingForTrust/tft/blob/master/userModel-densenet.py

3. The user must define the following variables in the python file.

   a. `PATH_TO_THE_MODEL` – Absolute path to the model
   b. `IMAGE_SAMPLES_FOLDER` – A folder containing test images
   c. `IMAGE_Vs_LABELS_CSV` – A csv file that has the mapping between image names `and corresponding labels. The column names of the csv file must be "ImageName"` and "Label" respectively.

ImageName	Label
n02085620_199.JPEG	Chihuahua
n02085620_368.JPEG	Chihuahua
n02086079_1759.JPEG	Pekinese
n02086079_1820.JPEG	Pekinese

   d. `PATH_TO_SAVE_RESULTS` – Must be the following path under tomcat webapps folder.

   `<`**`tomcat webapps folder`**`>\teachntest\assets\results`

   e. `PATH_TO_JSON_INDEX_CLASS_MAPPING` – A json file that has the mapping between the class ids and corresponding human readable names. For Example:

```
{
    "0": "others",
    "1": "tench",
    "2": "goldfish",
    "3": "great_white_shark",
    "4": "tiger_shark",
    "5": "hammerhead"
}
```

   f. `PROJECT_NAME` – A string which represents a name under which the test is performed.

   g. `model_file_name` – A string which is the file name along with the extension of the model under test. It must be one of **`.h5`**, **`.meta`** and **`.pb`** file names. This variable is a must while conducting Robustness test using adversarial patch method.

4. With #1, #2 and #3 in place, the developer can start auditing the model by instantiating the Model class and then passing it along with necessary variables to the respective auditing methods and calling a run() method for auditing.

   For example:

```
model = Model(PATH_TO_THE_MODEL, model_file_name))

method1_generalization = RotationTranslation(model,
IMAGE_SAMPLES_FOLDER,IMAGE_VS_LABELS_CSV,
PATH_TO_SAVE_RESULTS,PROJECT_NAME,
PATH_TO_JSON_INDEX_CLASS_MAPPING, threshold=0.1)
```

   `method1_generalization.run()` # returns a string message whether results generated successfully or not.

   Note:

   1. For testing with respect to methods under **AdversarialInputs**(i.e fast gradient sign method[fgsm] and carlini & wagner[cw] methods) and **ObjectRecognitionWeakSignals** (i.e. grayscale, contrast, additive noise and Eidolon) respective function names are named after the methods instead of run(). For example: to audit against these methods after instantiating **ObjectRecognitionWeakSignals** class, below depiction shows how functions need to be invoked.

```
audit1 = AdversarialInputs(model, IMAGE_SAMPLES_FOLDER, IMAGE_VS_LABELS_CSV, PATH_TO_SAVE_RESULTS, PROJECT_NAME,
                           PATH_TO_JSON_INDEX_CLASS_MAPPING, 4, threshold=0.1, targeted=True, target_class=2)

result = audit1.fgsm(0.015)  # provide epsilon value
print(result)

result = audit1.cw()  # takes optional learning rate & number of iterations.
print(result)
```

```
audit4 = ObjectRecognitionWeakSignals(model, IMAGE_SAMPLES_FOLDER, IMAGE_VS_LABELS_CSV, PATH_TO_SAVE_RESULTS,
                                      PATH_TO_JSON_INDEX_CLASS_MAPPING, threshold=0.1)

result = audit4.generate_gray_scale()
print(result)
result = audit4.generate_low_contrast(contrast_level_1=0.6)
print(result)
result = audit4.generate_noisy(noise_width=0.1, contrast_level_2=0.3)
print(result)

"""
- reach: float, controlling the strength of the manipulation
- coherence: a float within [0, 1] with 1 = full coherence
- grain: float, controlling how fine-grained the distortion is
"""
result = audit4.generate_eidolon(grain=10.0, coherence=1.0, reach=2.0)
print(result)
```

   2. A method under Robustness called AdversarialPatches has lot of parameters (including input tensor, logit tensor names & the model file name [.meta, .h5 or .pb]) to be provided to run as compared to other methods. We suggest referring further

details of it in documentation provided in **`robustness_image.py`** file [This file will be present under "tft" folder in site-packages of the anaconda environment where "tft" has been installed]
A code snippet to run it is as shown below.

```
model = Model(PATH_TO_THE_MODEL, CHECK_POINT)

audit0 = AdversarialPatches(model, IMAGE_SAMPLES_FOLDER, IMAGE_VS_LABELS_CSV, PATH_TO_SAVE_RESULTS, PROJECT_NAME,
                            PATH_TO_JSON_INDEX_CLASS_MAPPING, 1001, PATH_TO_THE_MODEL, model_file_name, 'x:0',
                            'InceptionV3/Logits/SpatialSqueeze:0', 13, (-1, 1), 'rectangle', 4)
print("\n\n results are :\n\n", audit0.run())
```

And attached herewith the links to user model files for inceptionV3 model, Mobile net Resnet50 & DenseNet models as reference.

InceptionV3:

https://github.com/testingForTrust/tft/blob/master/userModel-inceptionV3.py

MobileNet:

https://github.com/testingForTrust/tft/blob/master/userModel-mobilenet.py

Resnet50:

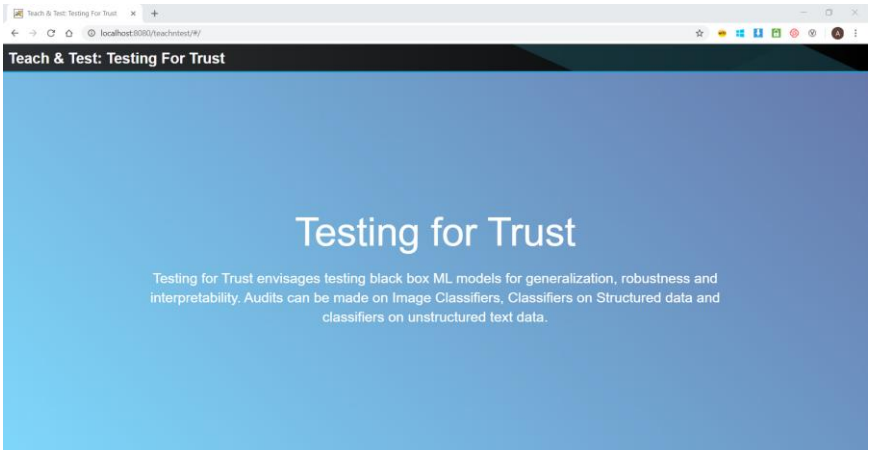
https://github.com/testingForTrust/tft/blob/master/userModel_resnet50.py

DenseNet:

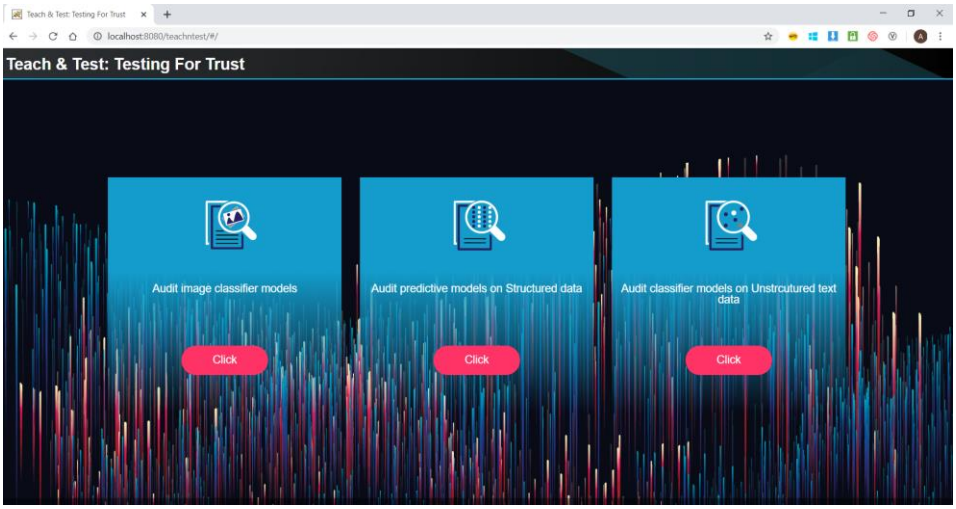
https://github.com/testingForTrust/tft/blob/master/userModel-densenet.py

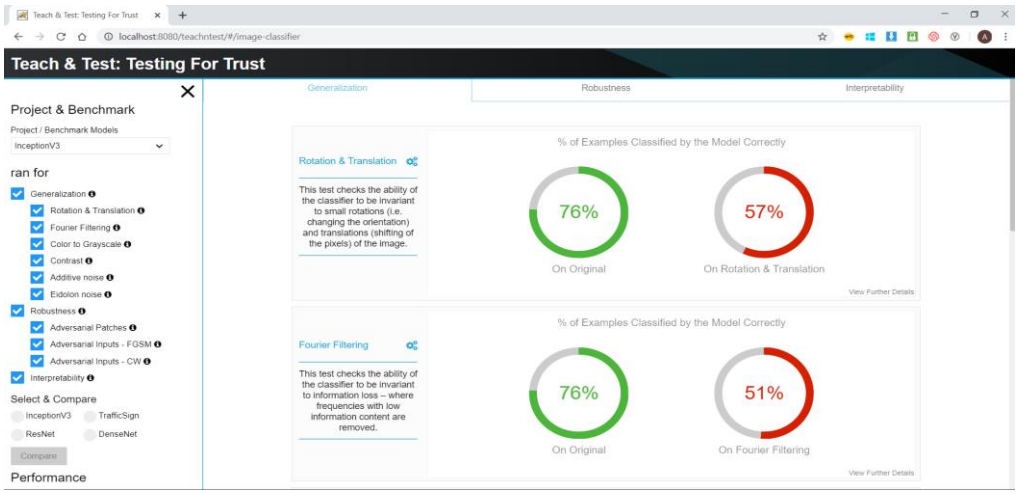
5. Open a command prompt, navigate to the path `<tomcat webapps folder>\teachntest\assets` and run the python file `flaskScript.py` (using the command `python flaskScript.py`) that helps in:
   a. ***selecting projects for which results to be visualized***
   b. ***compare the results with custom and benchmark models***
   c. ***capture the comments while interpreting the results***

6. To visualize the results, just start the tomcat server and type the URL `http://localhost:<port#>/teachntest` [port# = port number which tomcat was configured to listen to during installation. Better to have any ports other than 80, 8080]



7. **Scroll and click** on “Audit image classifier models” for visualizing results for different methods. Screenshots below.





8. There are results that have already been run for state-of-the-art models –

[i] InceptionV3, ResNet-50 and DenseNet run for 50 ImageNet classes.

[ii] MobileNet for 4 classes (4 traffic signs)

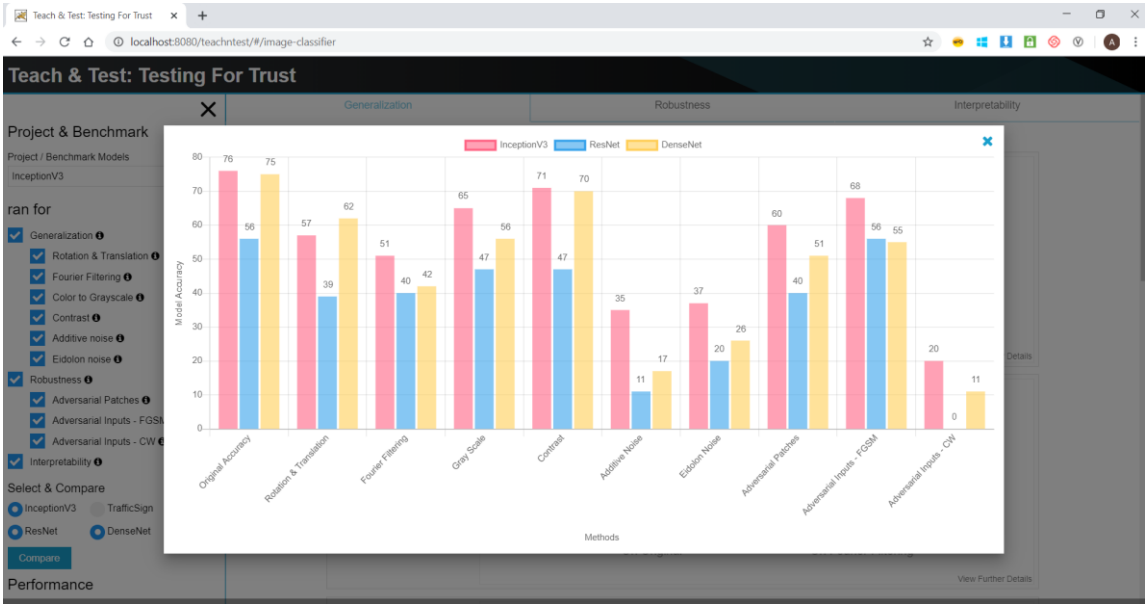
And the same is available at the following location on GoogleDrive since the total size is around 6GB.

https://drive.google.com/drive/folders/1yDArVWc7rlQrOfefAj8aByULYd_CTndt?usp=sharing

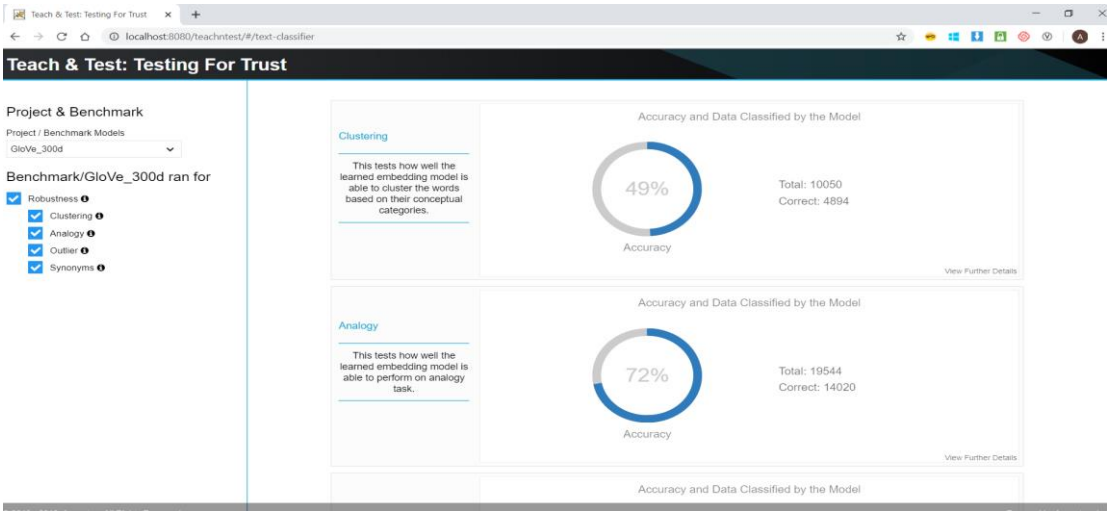
You would find four .zip files corresponding to DenseNet, InceptionV3, ResNet50 and MobileNet (TrafficSign Classifier) in the above location. One must download each of them, extract and place the extracted folder under the path:
<tomcat webapps folder>\teachntest\assets\results\Benchmark to compare a custom model with these benchmark models.

For **GloVe 300d** word embedding, the results have been included in "teachntest.zip" which can be seen through UI.

A depiction of the histogram chart showing the comparison of image classifier models.



9. On similar lines, word embeddings can also be tested. However:
   (a) so far we have worked with GloVe embedding models (50d, 100d, 200d & 300d word vectors) in ".txt" format that goes as input to TFT.
   (b) Integration of few other methods and some features are in progress & part of future work.



D. Performance Metrics of Image Classifier Models

Another aspect where this library comes useful is in evaluating the following:

[a] time to inference
[b] time it takes to pre_process a batch of data by the model
[c] time to load the model
[d] time spent on each layer by the model

This can be done by following the steps in usermodel file as below.

1. importing the class "`PerformanceMetrics`" like below:

```
from tft.performance_image.performance_metrics import PerformanceMetrics
```

2. instantiating an object of this class with necessary parameters as below:
   where "model" is the instance of the Model class defined in the usermodel file. Explanation for other parameters can be found in **performance_metrics.py** file under tft package.

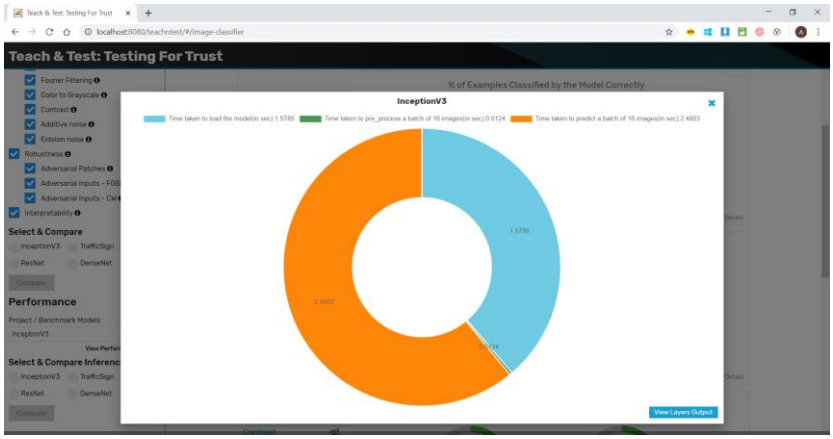
```
pm = PerformanceMetrics(model, "x:0", PATH_TO_THE_MODEL, model_file_name, True, PATH_TO_SAVE_RESULTS, PROJECT_NAME,
                        IMAGE_SAMPLES_FOLDER, IMAGE_VS_LABELS_CSV, model.image_size_height, model.image_size_width)
```

3. calling the compute() function as below:

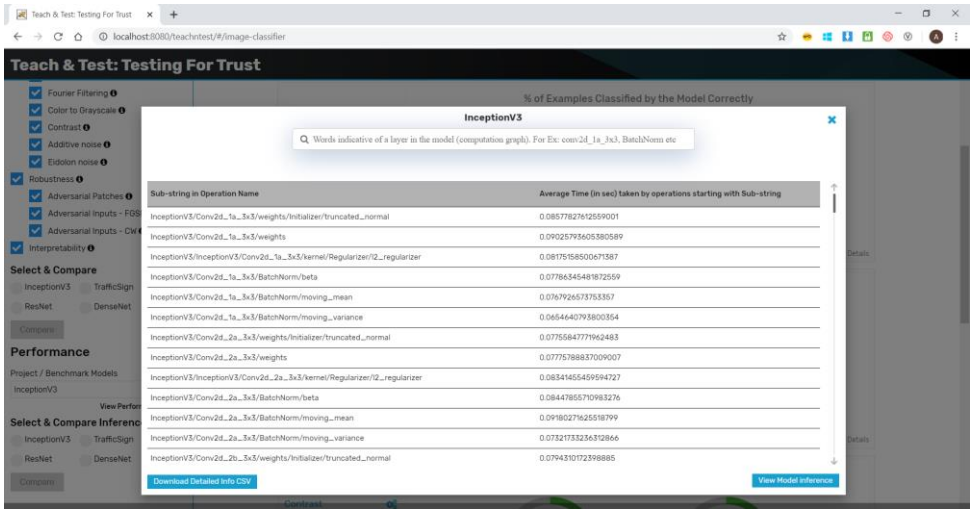
```
pm.compute()
```

   For reference, once can have a look at the userModel files prepared for InceptionV3, MobileNet, ResNet & Densenet as shared through the links in previous section.

4. In the UI, a doughnut graph shows following metrics.
   [a] time to inference
   [b] time it takes to pre_process a batch of data by the model
   [c] time to load the model



5. The layers' information can be seen through "View Layers Output" button in the doughnut chart. With the help of a search functionality, users could come to know about how much time has been spent in a layer (constituted by the operations in the computation graph). There is an **option** to quickly **download** the detailed information on time spent at each operation in a layer.



6. The custom model's inference can be compared with benchmark models as shown below.