Sample Applications

Sample Applications

Sample Applications Overview

The following applications demonstrate the use of the TensorFlow Lite for Microcontrollers framework with the Simplicity SDK.

Voice Control Light

This application demonstrates a neural network with TensorFlow Lite for Microcontrollers to detect the spoken words "on" and "off" from audio data recorded on the microphone in a Micrium OS kernel task.

The detected keywords are used to control an LED on the board. The audio data is sampled continuously and preprocessed using the Audio Feature Generator component. Inference is run every 200 ms on the past ~1 s of audio data.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

Z3SwitchWithVoice

This application combines voice detection with Zigbee 3.0 to create a voice-controlled switch node that can be used to toggle a light node. The application uses the same model as Voice Control Light to detect the spoken keywords "on" and "off". Upon detection, the switch node sends On/Off commands over the Zigbee network.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

TensorFlow Lite Micro - Hello World

This application demonstrates a model trained to replicate a sine function and use the inference results to fade an LED. The application is originally written by TensorFlow, but has been ported to the Simplicity SDK.

The model is approximately 2.5 KB. The entire application takes around 157 KB flash and 15 KB RAM. This application uses large amounts of flash memory because it does not manually specify which operations are used in the model and, as a result, compiles all kernel implementations.

The application illustrates a minimal inference application and serves as a good starting point for understanding the TensorFlow Lite for Microcontrollers model interpretation flow.

This sample application uses a fixed model contained in `hello_world_model_data.cc`.

TensorFlow Lite Micro - Micro Speech

This application demonstrates a 20 KB model trained to detect simple words from speech data recorded from a microphone. The application is originally written by TensorFlow, but has been ported to the Simplicity SDK.

This application uses around 100 KB flash and 37 KB of RAM. Around 10 KB of the RAM usage is related to FFT frontend and to store audio data. With a clock speed of 38.4 MHz and using the optimized kernel implementations, the inference time on ~1 s of audio data is approximately 111 ms.

This application illustrates the process of generating features from audio data and doing detections in real time. It also demonstrates how to manually specify which operations are used in the network, which saves a significant amount of flash.

This sample application uses a fixed model contained in `micro_speech_model_data.cc`.

TensorFlow Lite Micro - Magic Wand

This application demonstrates a 10 KB model trained to recognize various hand gestures using an accelerometer to detect the motion. The detected gestures are printed to the serial port. The application is originally written by TensorFlow, but has been ported to the Simplicity SDK.

This application uses around 104 KB flash and 25 KB of RAM. This application demonstrates how to use accelerometer data as inference input and also shows how to manually specify which operations are used in the network, which saves a significant amount of flash.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

TensorFlow Model Profiler

This application is designed to profile a TensorFlow Lite Micro model on Silicon Labs hardware. The model used by the application is provided by a TensorFlow Lite flatbuffer file called model.tflite in the config/tflite subdirectory. The profiler will measure the number of CPU clock cycles and elapsed time in each layer of the model when performing an inference. It will also produce a summary when inference is done. The input layer of the model is filled with all zeroes before performing a single inference. Profiling results are transmitted over VCOM.

To run the application with a different .tflite model, you can replace the file called model.tflite with a new TensorFlow Lite Micro flatbuffer file. This new file must also be called "model.tflite" and be placed inside the config/tflite subdirectory to be picked up by the sample application. After the model has been replaced, regenerate the project.

To load and perform inference on a TensorFlow Lite Micro model, allocate a number of bytes to a "tensor arena" to hold state needed by the TensorFlow Lite Micro. The size of this tensor arena depends on the size of the model and the number of operators. The TensorFlow Model Profiler application can be used to measure the amount of RAM needed by the tensor arena to load the specific TensorFlow Lite Micro model. This is measured by dynamically allocating RAM for the tensor arena and reporting the number of bytes needed on VCOM. The number of bytes needed for the tensor arena can later be used to statically allocate memory when the model is used in a different application.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

Rock-Paper-Scissors (Image Classification)

Image classification is one of the most important applications of deep learning and Artificial Intelligence. Image classification refers to assigning labels to images based on certain characteristics or features present in them. The algorithm identifies these features and uses them to differentiate between different images and assign labels to them.

This application uses TensorFlow Lite for Microcontrollers to run image classification machine learning models to classify hand gestures from image data captured from ArduCAM camera. The detection is visualized using the LED's on the board and the classification results are written to the VCOM serial port.

For more information, refer to the Image Classifier documentation.

Image Classifier

Image Classifier

Image classification is one of the most important applications of deep learning and Artificial Intelligence. Image classification refers to assigning labels to images based on certain characteristics or features present in them. The algorithm identifies these features and uses them to differentiate between different images and assign labels to them.

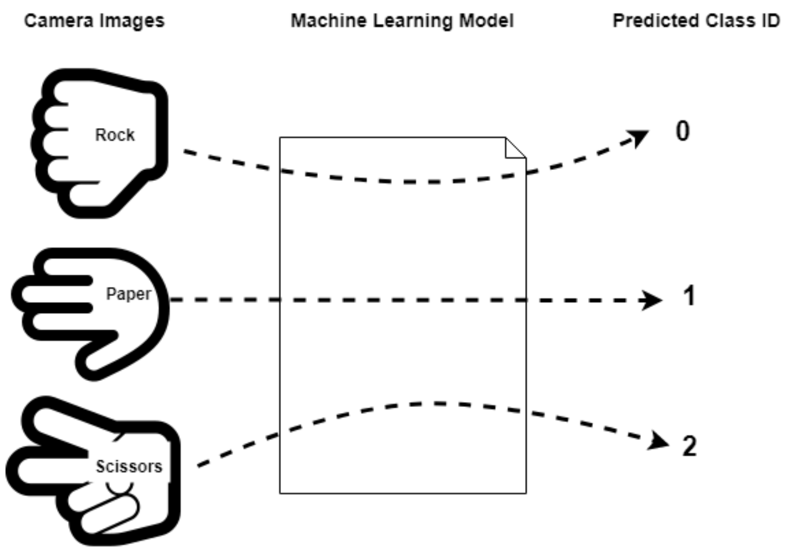
This application uses TensorFlow Lite for Microcontrollers to run image classification machine learning models to classify hand gestures from image data captured from ArduCAM camera. The detection is visualized using the LED's on the board and the classification results are written to the VCOM serial port.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

Class Labels

- rock: Images of a person's hand making a “rock” gesture
- paper: Images of a person's hand making a “paper” gesture
- scissors: Images of a person's hand making a “scissors” gesture
- unknown: Random images not containing any of the above

Following figure describes class labels with respect to hand pose.



Required Hardware and Setup

- Silicon Labs EFR series boards BRD2601B or BRD2608A.
- Berg Strip Connectors with Jumper wires (minimum 8 count) any one of following combination.
  - Male-Berg strip with female to female jumper wires.
  - Female-Berg strip with male to female jumper wires.
- ArduCAM camera module.

Pin Configuration

Following table shows pin connections between ArduCAM and Development kit.

ArduCAM Pin	Board Expansion Header Pin
GND	1
VCC	18
CS	10
MOSI	4
MISO	6
SCK	8
SDA	16
SCL	15

Required Software

- Simplicity studio v5 with
  - simplicity_sdk
  - aiml-extension

Dumping Images to PC

This application uses JLink to stream image data to a Python script located at aiml-extension/tools/image-visualization. To get started, refer to the instructions provided in the readme.md file. Once the application binary is flashed onto the development board, you can launch the visualization tool to view incoming image streams and optionally save them to your local PC.

Application Notes

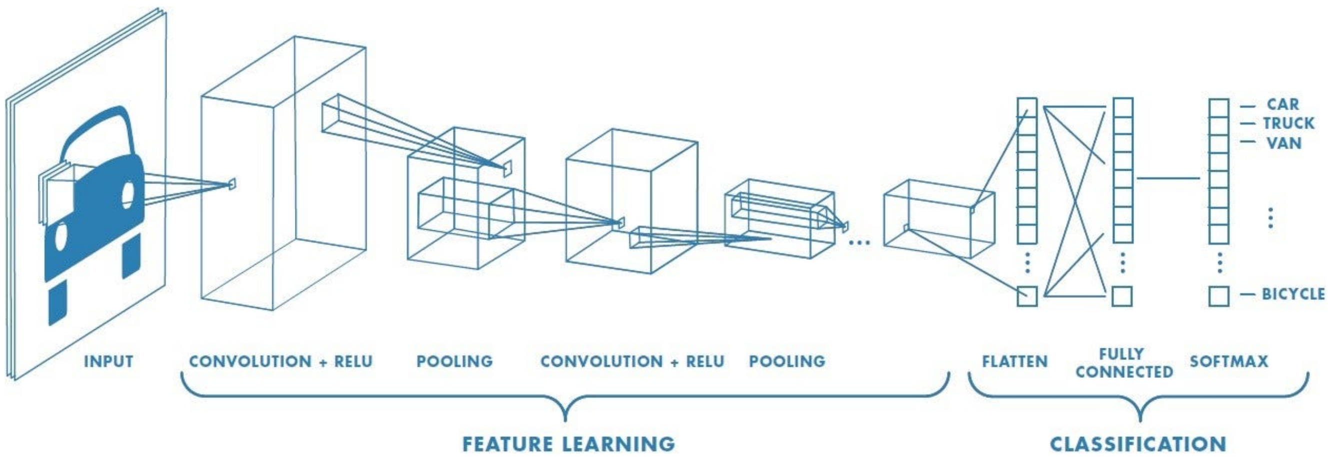
- Camera Input: This application utilizes images captured using an ArduCAM camera.
- Lighting Conditions: The model was trained on well-lit images. To achieve optimal results, conduct experiments in good lighting. Adjust the SL_ML_IMAGE_MEAN_THRESHOLD parameter in the .slcp configuration file to set the minimum mean intensity required for image processing.
- Recommended Distance: Maintain approximately 0.5 meters between the camera and the subject's hand during experimentation.
- Background Setup: Use a plain background (preferably white or black) to improve detection accuracy and consistency.

Convolutional Neural Networks

The type of machine learning model used in this application is Convolutional Neural Network (CNN).

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.

A typical CNN can be visualized as follows:



A typical CNN is comprised of multiple layers. A given layer is basically a mathematical operation that operates on multi-dimensional arrays (a.k.a tensors). The layers of a CNN can be split into two core phases:

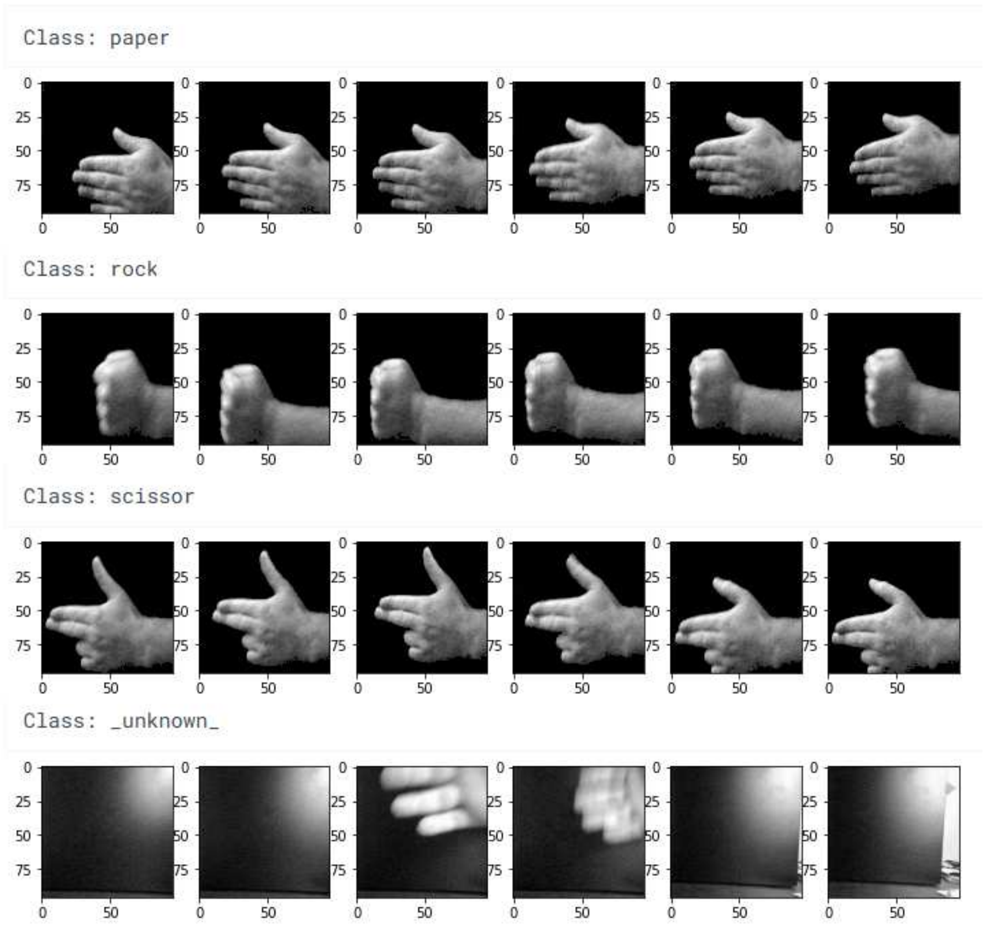
- Feature Learning: This uses Convolutional layers to extract "features" from the input image
- Classification: This takes the flatten "feature vector" from the feature learning layers and uses "fully connected" layer(s) to make a prediction on which class the input image belongs

Deep Learning Pipeline

A deep learning pipeline typically consists of three primary stages: dataset collection, model training, and inference as follows.

Dataset
Training
Inference

Some sample data used for training the model in this application is illustrated in the figure below.



```
Class: paper
Class: rock
Class: scissor
Class: _unknown_
```

Data Preprocessing

The preprocessing stage involves specific data normalization settings, managed by the ml_image_feature_generation component.

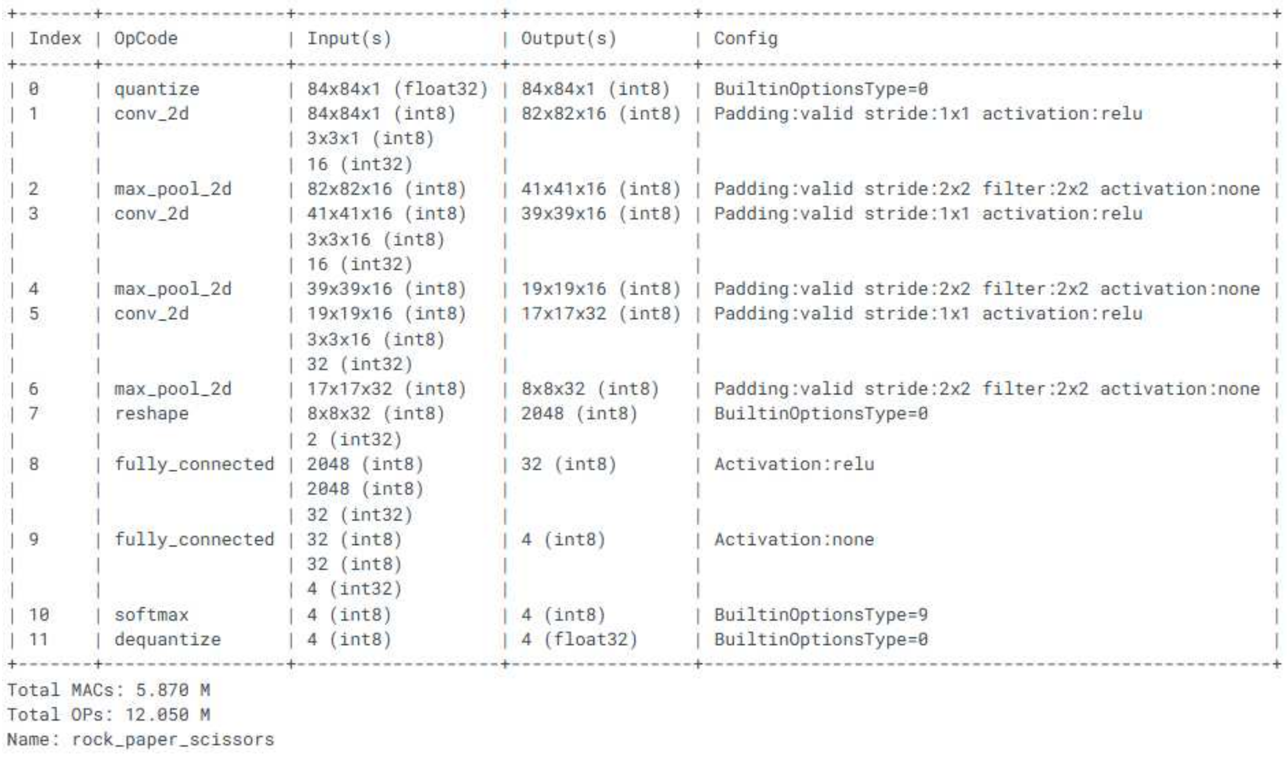
samplewise_center = True samplewise_std_normalization = True

norm_img = (img - mean(img)) / std(img)

These settings normalize each input image individually using the formula. This helps to ensure the model is not as dependent on camera and lighting variations.

Model Details

The model summary presents detailed information about each layer along with the number of parameters involved. This breakdown outlines the architecture used to perform image classification task.



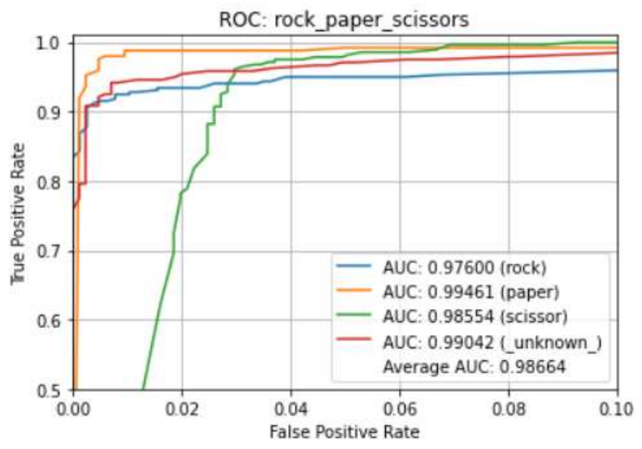
```
+-------+-----------------+------------------+-----------------+-------------------------------------------------------+
| Index | OpCode          | Input(s)         | Output(s)       | Config                                                |
+-------+-----------------+------------------+-----------------+-------------------------------------------------------+
| 0     | quantize        | 84x84x1 (float32) | 84x84x1 (int8)  | BuiltinOptionsType=0                                  |
| 1     | conv_2d         | 84x84x1 (int8)   | 82x82x16 (int8) | Padding:valid stride:1x1 activation:relu              |
|       |                 | 3x3x1 (int8)     |                 |                                                       |
|       |                 | 16 (int32)       |                 |                                                       |
| 2     | max_pool_2d     | 82x82x16 (int8)  | 41x41x16 (int8) | Padding:valid stride:2x2 filter:2x2 activation:none |
| 3     | conv_2d         | 41x41x16 (int8)  | 39x39x16 (int8) | Padding:valid stride:1x1 activation:relu              |
|       |                 | 3x3x16 (int8)    |                 |                                                       |
|       |                 | 16 (int32)       |                 |                                                       |
| 4     | max_pool_2d     | 39x39x16 (int8)  | 19x19x16 (int8) | Padding:valid stride:2x2 filter:2x2 activation:none |
| 5     | conv_2d         | 19x19x16 (int8)  | 17x17x32 (int8) | Padding:valid stride:1x1 activation:relu              |
|       |                 | 3x3x16 (int8)    |                 |                                                       |
|       |                 | 32 (int32)       |                 |                                                       |
| 6     | max_pool_2d     | 17x17x32 (int8)  | 8x8x32 (int8)   | Padding:valid stride:2x2 filter:2x2 activation:none |
| 7     | reshape         | 8x8x32 (int8)    | 2048 (int8)     | BuiltinOptionsType=0                                  |
|       |                 | 2 (int32)        |                 |                                                       |
| 8     | fully_connected | 2048 (int8)      | 32 (int8)       | Activation:relu                                       |
|       |                 | 2048 (int8)      |                 |                                                       |
|       |                 | 32 (int32)       |                 |                                                       |
| 9     | fully_connected | 32 (int8)        | 4 (int8)        | Activation:none                                       |
|       |                 | 32 (int8)        |                 |                                                       |
|       |                 | 4 (int32)        |                 |                                                       |
| 10    | softmax         | 4 (int8)         | 4 (int8)        | BuiltinOptionsType=9                                  |
| 11    | dequantize      | 4 (int8)         | 4 (float32)     | BuiltinOptionsType=0                                  |
+-------+-----------------+------------------+-----------------+-------------------------------------------------------+
Total MACs: 5.870 M
Total OPs: 12.050 M
Name: rock_paper_scissors
```

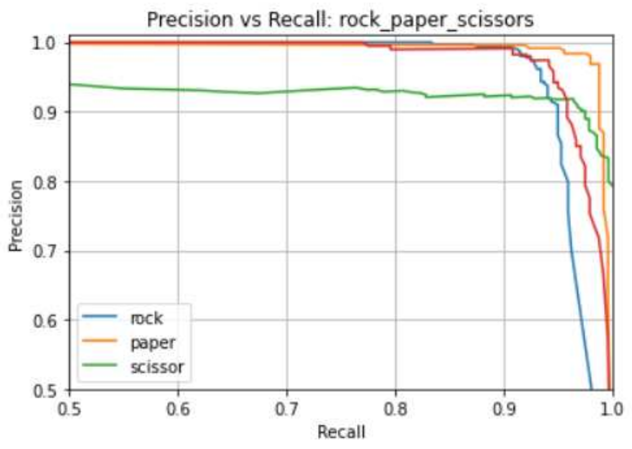
Model Evaluation

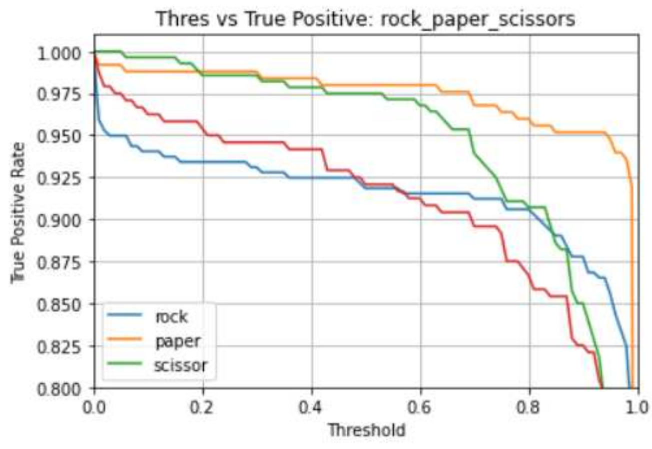
The fundamental approach to model evaluation involves feeding test samples—new, unseen data not used during training—into the model and comparing its predictions against the actual expected values. If every prediction is correct, the model achieves 100% accuracy; each incorrect prediction lowers the overall accuracy.

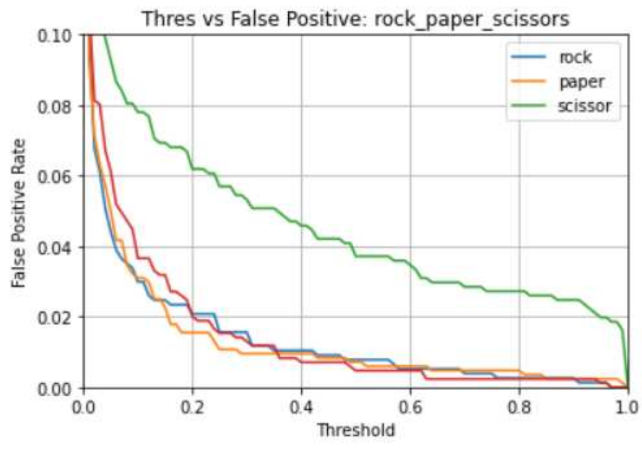
The figure below illustrates key performance indicators (KPIs) achieved using the Rock-Paper-Scissors dataset, including:

Accuracy: Proportion of correct predictions over total predictions ROC Curve: Graphical representation of true positive rate vs. false positive rate Recall: Measure of the model’s ability to identify all relevant instances Precision: Proportion of true positives among all predicted positives









Overall accuracy: 95.037% Class accuracies:

- paper = 98.394%
- scissor = 97.500%
- rock = 92.476%
- unknown = 92.083% Average ROC AUC: 98.664% Class ROC AUC:
- paper = 99.461%
- unknown = 99.042%
- scissor = 98.554%
- rock = 97.600%