## Files Submitted

### Submission Files

The project submission includes all required files.

- Ipython notebook with code
- HTML output of the code
- A writeup report (either pdf or markdown)

## Dataset Exploration

### Dataset Summary

The submission includes a basic summary of the data set.

Dataset contained three different datasets:

- train.p -> this file contains the dataset, which is used for training a neural network model.
- valid.p -> this dataset is used for validating the model trough training steps.
- test.p -> after the training step this dataset is used to test the trained model.

### Exploratory Visualization

The submission includes an exploratory visualization on the dataset.

As we can see in executed source code.

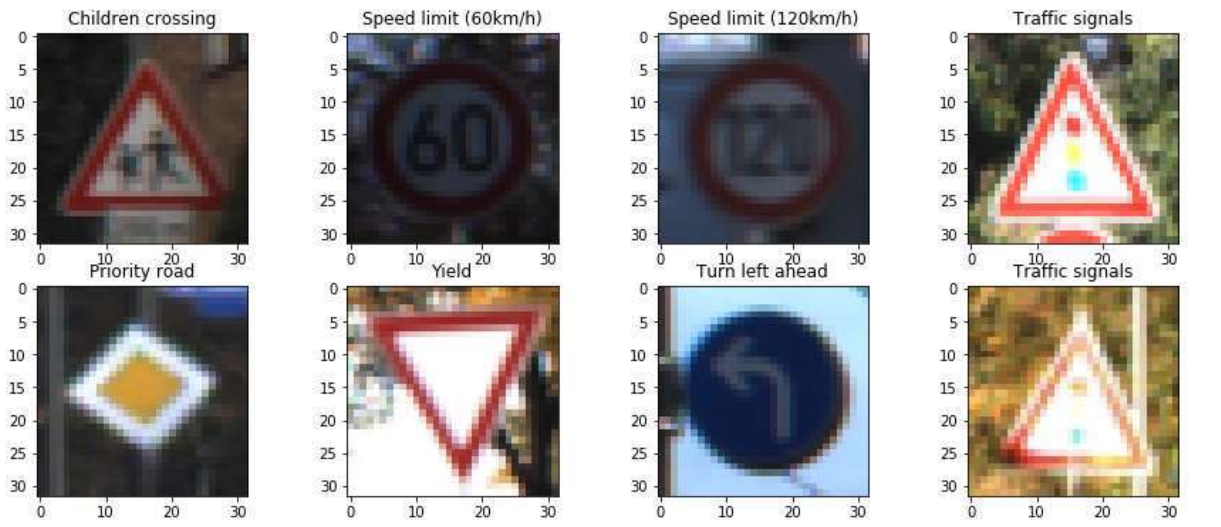
```
Number of training examples = 34799
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Note: the number of classes is 43 because the train, validate and test dataset only contain 43 different German traffic signs.

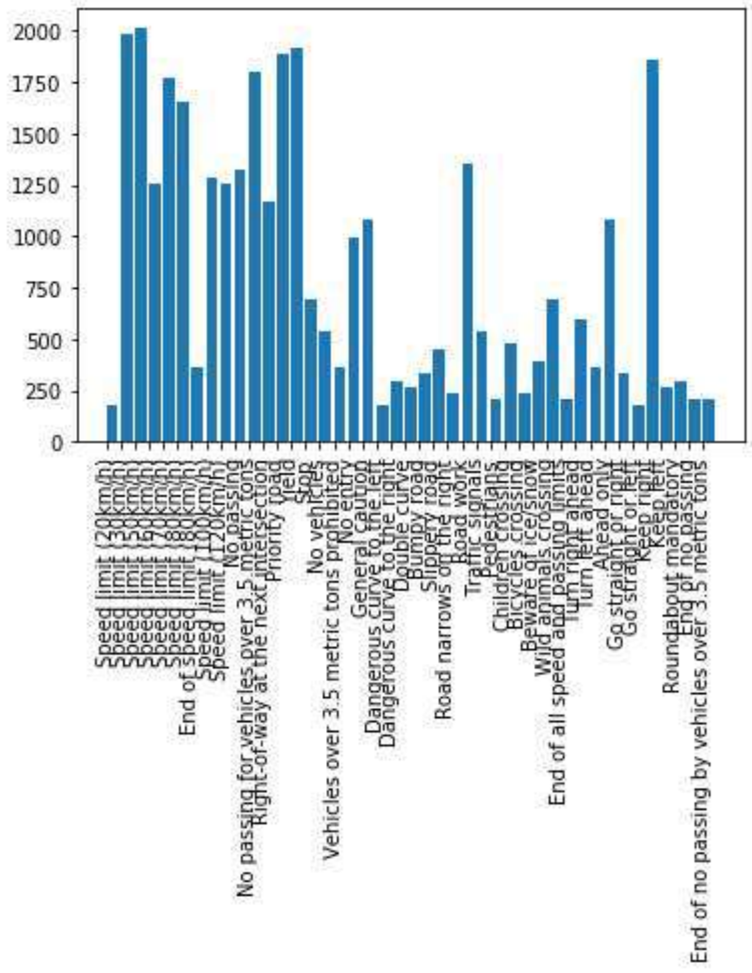
For this project we use only two columns of datasets: features and labels

- Features: contains the images (32x32x3)
- Labels: contains the Id of traffic sign

Each row of features a row in labels which determines the traffic sign classification.



For the name of each traffic sign class the project contains ‘signnames.csv’ file which is used in above example to display the name of traffic sign and the next figure visualizes the train dataset.



## Design and Test a Model Architecture

### Preprocessing

| The submission describes the preprocessing techniques used and why these techniques were chosen. |
|---|

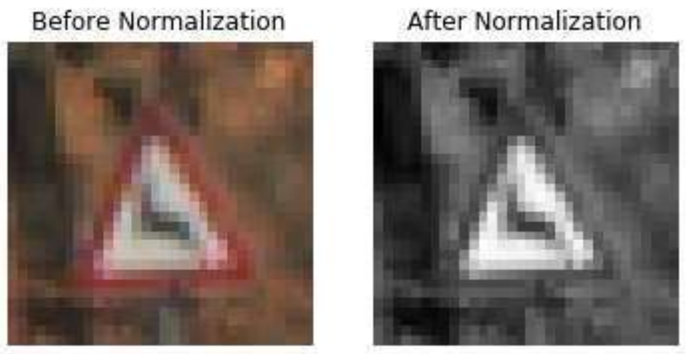
For the reprocessing the project’s description suggested the following method:

```
def normalizationI(img):
   return (img - 128)  / 128
```

But after testing this function I noticed the performance of the training was not very good and I decided to consider the gray scale and change the 3-channel image to 1 channel therefor I developed the following function:

```
def normalizationII(img):
   img = 0.299 * img[:, :, 0] + 0.587 * img[:, :, 1] + 0.114 * img[:, :, 2]
   img = (img / 255.).astype(np.float32)
   return img.reshape(32,32,1)
```

The performance of training was better with this function. Furthermore, because the shape of image after normalization is (32x32) I have reshaped the image to (32x32x1) to be valid for next step. Below we have a figure which demonstrates a traffic sign before and after normalization.



## Model Architecture

| |
|---|
| The submission provides details of the characteristics and qualities of the architecture, including the type of model used, the number of layers, and the size of each layer. Visualizations emphasizing particular qualities of the architecture are encouraged. |

In the following I’ll explain about each layer of LENET model. This model includes 5 layers. The final model architecture consists the following layers.

| | |
|---|---|
| Input | Input = (32x32x1) |

| | |
|---|---|
| Convolutional Layer | Strides = 1x1<br>Padding = Valid<br>Filter size = 5x5<br>Filter depth = 6<br>Output = (28x28x6) |
| For initializing the weights at the beginning I have used tf.truncated_normal function. | |

Initializing the weights with random number from a normal distribution is a good practice because randomizing the weights helps the model from becoming stuck in the same place every time it trains.
Choosing weights from a normal distribution prevents any one weights from overwhelming other weights. The tf.truncated_normal method generates random numbers from a normal distribution and the return value is a tensor, which will be used be use in the first convolutional layer as follows.

```
conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))
conv1_b = tf.Variable(tf.zeros(6))
conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b
```

For the biases I have used the tf.zeros method because the weights are random and we don't need to randomize the bias. The output of this method is a tensor and is used as the above code line.

| RELU | |
|---|---|

The Rectified Linear method (RELU) is a type of non-Linear activation method and this method converts some values to zero and the training logic changes to non-linear method. This method returns 0 if x is negative, otherwise it returns x. To use the RELU functionality we can use the tf.nn.relu method as follows.

```
conv1 = tf.nn.relu(conv1)
```

The RELU method effectively turns off the negative weights.

| Dropout | Keen_prob = 0.7 |
|---|---|

The dropout reduces the value of some random selected weights to zero, making null the subsequent layers.
This method avoids all the neurons converging to the same goals and decorates the adapted weights.
Because of this regularization we can train much larger models. TensorFlow has the tf.nn.dropout method for regularization and it can be used as in this code snippet.

```
conv1 = tf.nn.dropout(conv1 , 0.7)
```

The second parameter in the above code line is keep_prob and this is the probability of keeping a neuron.

| Max pooling | Input = (28x28x6)<br>Output = (14x14x6) |
|---|---|

Max pooling reduces the size of the input therefore, the neural network will focus on the most important elements. Max pooling causes that the training be done on maximum values of each filtered layer and the other values are removed. Max pooling is used to decrease the size of output. We can see the input size is (28x28x6) but the output size is (14x14x6) and consequently the overfitting will be prevented. This is the Max pooling code line in my code in cell 9.

| onv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID') |
|---|

Since adding hidden layers to a network allows it to model more complex functions. The previous steps were the first layer but because it's not observable for outside the network, it's known as hidden layer.

| Convolutional Layer | Strides = 1x1<br>Padding = Valid<br>Filter size = 5x5<br>Filter depth = 16<br>Output = (10x10x6) |
|---|---|
| For the explanation please refer to RELU at abover.<br><br>conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev = sigma))<br>conv2_b = tf.Variable(tf.zeros(16))<br>conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b | |

| RELU | |
|---|---|
| It has been explained in the RELU cell above.<br>conv2 = tf.nn.relu(conv2) | |

| Dropout | Keen_prob = 0.7 |
|---|---|
| It has been explained in the Dropout cell above.<br>conv2 = tf.nn.dropout(conv2 , 0.7)<br>I have used the both regularizers in my code because I wanted to see test them together. | |

| Max pooling | Input = (10x10x16)<br>Output = (5x5x16) |
|---|---|
| It has been explained in the Max pooling cell above.<br>conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID') | |

| Fully connected Layer | Input = 400, Output=120 |
|---|---|
| The fully connected layer is different and is non-convolutional layer. All inputs are connected to all output neurons. This is referred to as a dense layer.<br>fc0   = flatten(conv2)<br>fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))<br>fc1_b = tf.Variable(tf.zeros(120))<br>fc1   = tf.matmul(fc0, fc1_W) + fc1_b | |

| RELU | |
|---|---|
| It has been explained in the RELU cell above.<br>fc1    = tf.nn.relu(fc1) | |

| Dropout | Keen_prob = 0.7 |
|---|---|
| It has been explained in the Dropout cell above.<br>fc1 = tf.nn.dropout(fc1 , 0.7) | |

| Fully connected Layer | Input = (120), Output = (84) |
|---|---|
| fc2_W = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))<br>fc2_b = tf.Variable(tf.zeros(84))<br>fc2 = tf.matmul(fc1, fc2_W) + fc2_b | |

| RELU | |
|---|---|
| It has been explained in the RELU cell above.<br>fc2 = tf.nn.relu(fc2) | |

| Dropout | Keen_prob = 0.7 |
|---|---|
| It has been explained in the Dropout cell above.<br>fc2 = tf.nn.dropout(fc2 , 0.7) | |

| Fully connected Layer | Input = (84), Output = (43) |
|---|---|
| fc3_W = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu, stddev = sigma))<br>fc3_b = tf.Variable(tf.zeros(43))<br>logits = tf.matmul(fc2, fc3_W) + fc3_b | |

| Output | Output = (43) |
|---|---|

## Model Training

| The submission describes how the model was trained by discussing what optimizer was used, batch size, number of epochs and values for hyperparameters. |
|---|

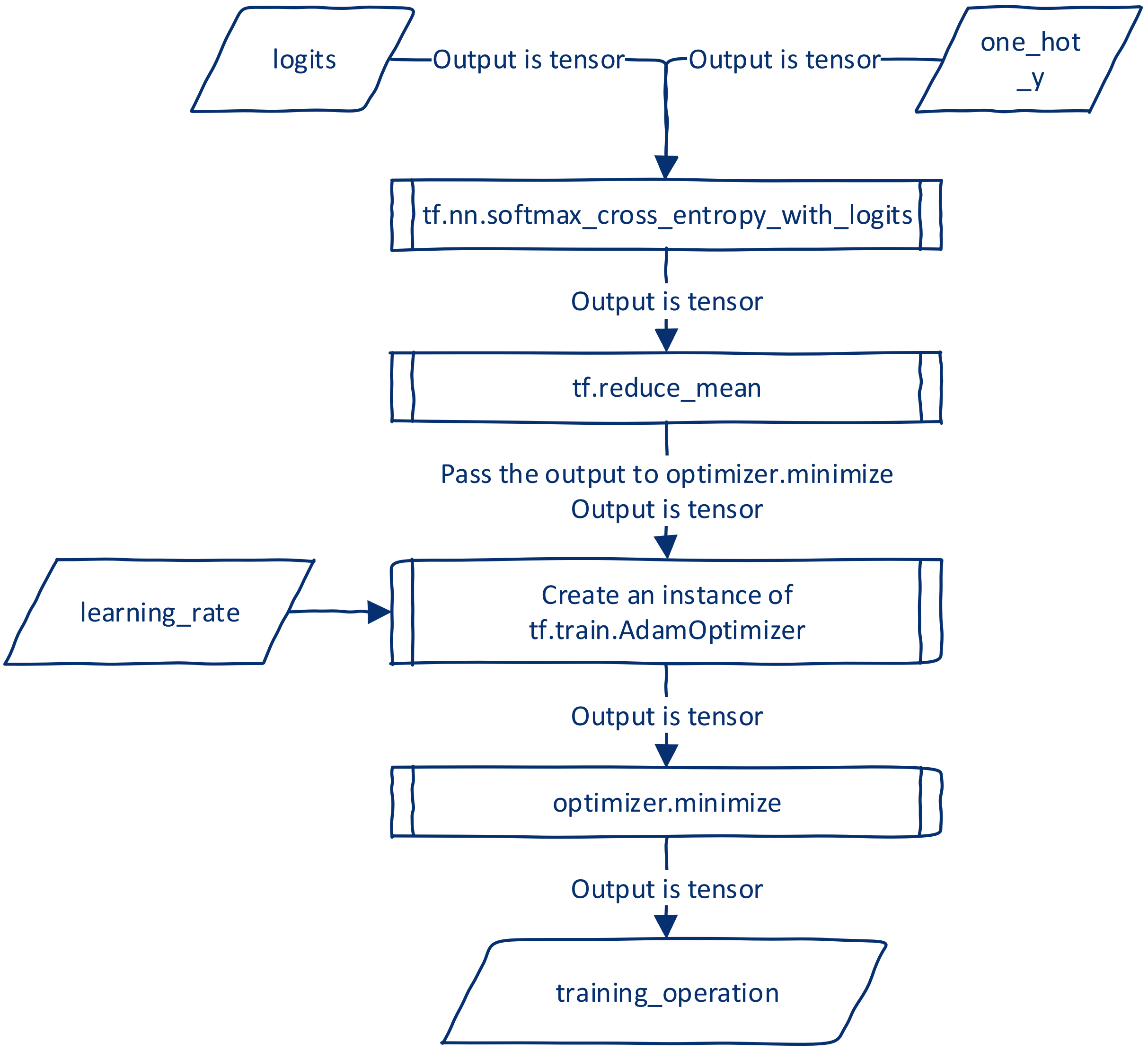
In cell 10 I have defined the EPOCHS variable and assigned 80 to it. It defines the number of iterations that the model will be trained as shown in cell 13. In general, more epochs the better model will train but the training takes longer.

In cell 11 I have defined the BATCH_SIZE variable and assigned 128 to it. It defines the number of images that must be run through the network at a time. The larger the batch size the faster model will train.

In cell 13 the learning rate variable rate has been defined with the assigned value 0.001. As we know lowering the learning rate would require more epochs but could ultimately achieve better accuracy.

The model is optimized via minimize optimizer of AdamOptimizer. In the following figure the flow chart of creating the minimize tensor is demonstrated.

Typically, the softmax functions are for classification and in this example I have used tf.nn.softmax_cross_entropy_with_logits Function. The next step is for calculating cost or loss I have used tf.reduce_mean as cross entropy error function. Finally, we should minimize the loss. Tensorflow has different loss optimization methods like f.train.GradientDescentOptimizer, tf.train.AdagradOptimizer, tf.train.AdadeltaOptimizer, tf.train.AdamOptimizertf. I have decided to use tf.train.AdamOptimizer. It stands for Adaptive Moment Estimation. This method considers two items first the average of the past gradient and second momentum factor. According to these factors it adds a factor when calculating gradients.



The code snippet for the above diagram is as follows.

```
'''
Define loss and optimizer
'''
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
```

```
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

### Solution Approach

The submission describes the approach to finding a solution. Accuracy on the validation set is 0.93 or greater.

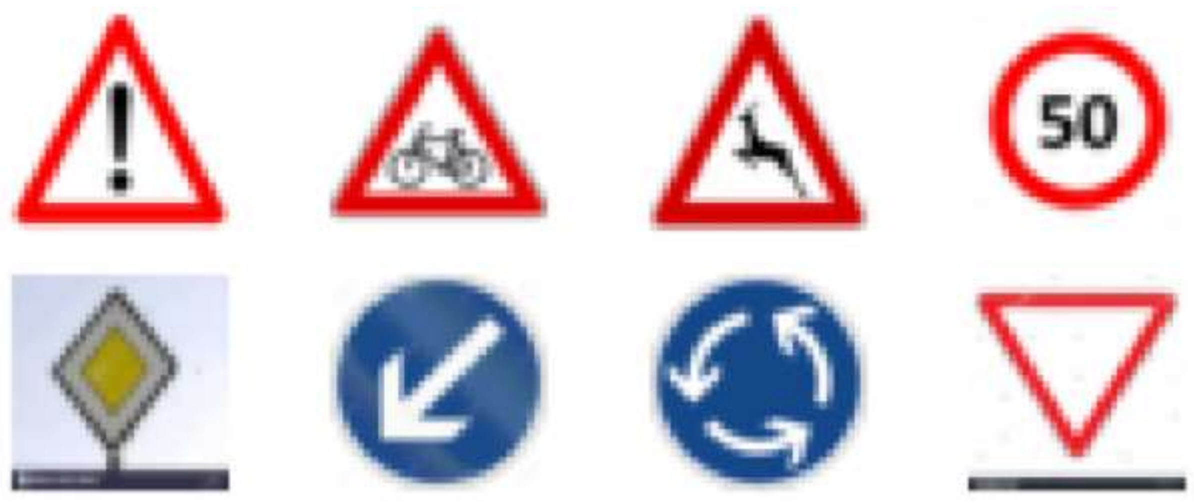
As you see in cell 35 of the submitted solution the accuracy of the validation is 100% and the accuracy of the test is 90.88% in cell 42. The **Figure 1** is the visualization of testing the model on new images.

## Test a Model on New Images

### Acquiring New Images

The submission includes five new German Traffic signs found on the web, and the images are visualized. Discussion is made as to particular qualities of the images or traffic signs in the images that are of interest, such as whether they would be difficult for the model to classify.

To test the accuracy of the model I have found eight traffic signs on the internet in the next figure I have visualizes them. It’s located in cell 33 in my Jupyter notebook.



The sample traffic signs in my project didn’t have same size therefore I have used the following code to have the same size.

```
image = scipy.misc.imresize(image, (32, 32))
```

### Performance on New Images

The submission documents the performance of the model when tested on the captured images. The performance on the new images is compared to the accuracy results of the test set.

As we can see below the test accuracy of the model was about 90,88% and the accuracy of model for my sample images was 100.00%.

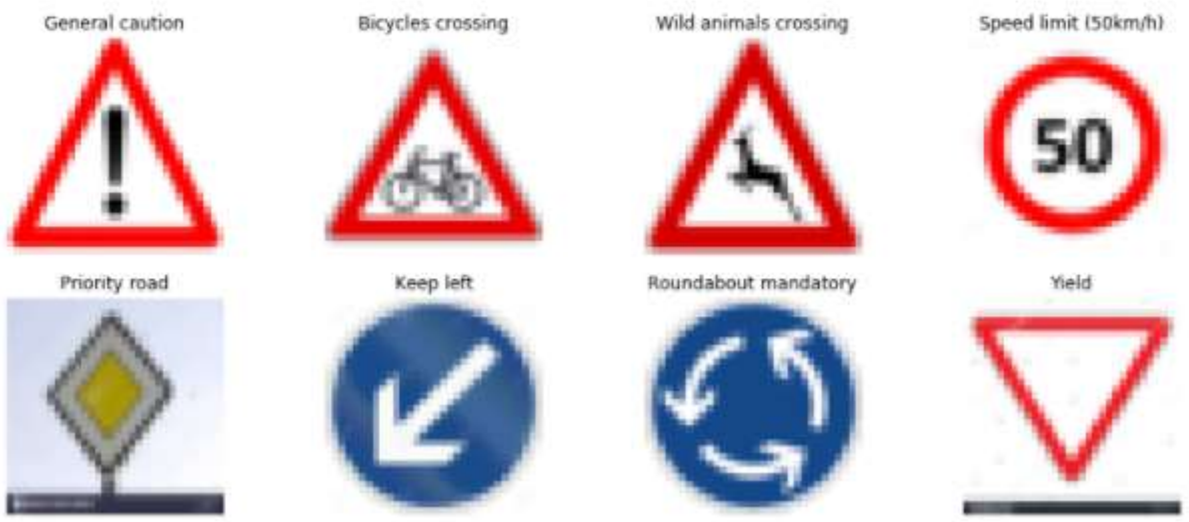


*Figure 1: Visualization of testing the model on new images.*

## Model Certainty - Softmax Probabilities

The top five softmax probabilities of the predictions on the captured images are outputted. The submission discusses how certain or uncertain the model is of its predictions.

If we look at the probabilities of Softmax in cell 36 we can notice if the model is detecting a sign correctly the certainty is about 99.8% to 100%. In my opinion if the sign is not detected or classified correctly or the model is not certain about it, it's due to the unclear interface of the samples that the model is trained according to them.

```
        == Actual ==          - General caution
   1.00000000000000000000% - General caution
   0.00000000000000000000% - Speed limit (20km/h)
   0.00000000000000000000% - Speed limit (30km/h)
   0.00000000000000000000% - Speed limit (50km/h)
   0.00000000000000000000% - Speed limit (60km/h)
```

```
        == Actual ==          - Bicycles crossing
   1.00000000000000000000% - Bicycles crossing
   0.00000000007160749771% - Children crossing
   0.00000000007122201440% - Bumpy road
   0.00000000000000000031% - Slippery road
   0.00000000000000000000% - No passing for vehicles over 3.5 metric to
ns
```

```
        == Actual ==          - Wild animals crossing
   0.99818998575210571289% - Wild animals crossing
   0.00179940566886216402% - Slippery road
   0.00000988879219221417% - Double curve
   0.00000076525617487277% - Dangerous curve to the left
```

```
0.00000000000000541028% - Bicycles crossing
```

```
     == Actual ==        - Speed limit (50km/h)
0.99999880790710044922% - Speed limit (50km/h)
0.00000014802219538979% - Speed limit (60km/h)
0.00000000002216930285% - Speed limit (30km/h)
0.00000000000043889108% - Keep right
0.00000000000000004620% - Speed limit (80km/h)
```