UPRM Hackathon Resources

Overview of Topics

Custom Image Dataset

Using Dataset to create a custom image dataset

Instead of using a pre-made dataset, we created our own custom image dataset for this Hackathon, using images from a NASA post on data.gov. You can also find the image files on Zenodo.

In order to create a custom image dataset, you have to start by loading the data in. The steps:

- Import the torch.utils.data.Dataset data primitive
- Define CustomImageDataset class
  - When defining it, you **must** pass Dataset as an argument
  - When defining it, you **must** implement at least these 3 functions:
    - **init**
    - **len**
    - **getitem**
- Store your dataset using the newly-defined CustomImageDataset class

Additional resources:

- You can find details and documentation on creating your own image dataset using PyTorch at WRITING CUSTOM DATASETS, DATALOADERS AND TRANSFORMS
- You can find an example walkthrough on Custom dataset in Pytorch —Part 1. Images

Using DataLoader on a custom image dataset

After you use the torch.utils.data.Dataset data primitive to store your dataset, you can use the torch.utils.data.DataLoader data primitive to iterate through it. DataLoader wraps an iterable around Dataset so that you can load your data into a model one batch at a time. Using this primitive allows you to train your model on a large dataset without needing to write a lot of code for batching your data. It's as simple as:

- Importing the torch.utils.data.DataLoader data primitive
- Creating a DataLoader object on your data
- Iterating through the DataLoader object

Here's an example from Building Custom Image Datasets in PyTorch:

```
import torch
import torchvision

mnist_train =
torchvision.datasets.MNIST('path/to/mnist_root/',train=True)

train_data_loader = torch.utils.data.DataLoader(mnist_train,
batch_size=32, shuffle=True, num_workers=16)

for batch_idx, batch in enumerate(train_data_loader):
    #inside this loop we can do stuff with a batch,
     like use it to train a model
```

Note that you must create a separate DataLoader for your train and test sets! If you want to follow the above code to create a DataLoader object for your test data, you must set your dataset's train parameter to False and then load that test dataset into your new DataLoader.

Also note that when setting the num_workers parameter in your DataLoader object to *n*, you dictate that *n* parallel processes will load batches simultaneously.

Classification Models

There is a plethora of machine learning models used for answering many different types questions. The one we're using is a classification model.

In image classification, a picture is "classified" as something. For example, an image of a cat could be classified as a cat, a dog, or even a bird. Obviously, we want our model to classify an image of a cat as a member of the cat class. How de we train our models to be able to do this?

To start, we feed our model a bunch of images and tell the model which class each image belongs to. Then, we feed our model new images and ask the model to predict which class each of these new images belongs to. If the model does well, we can use it on our data! If not, we want to retrain it to get higher prediction accuracy.

Here are some resources for further exploration:

- Does Image Classification Work? gives a great explanation of how image classification works and different machine learning methods implemented to classify images.
- Image Classification gives a short description and documents different types of classification models used for image classification on various datasets.

CNN

General

If you are interested in learning more about CNNs, here are some quick and easy resources!

- DEEPLIZARD Demo shows how a CNN works.

- What is a Convolutional Neural Network? A Beginner's Tutorial for Machine Learning and Deep Learning details an introductory explanation of CNNs
- Introduction to Convolutional Neural Networks gives further detail on all the components of a CNN model

Components

Input Layer – as you might expect, this layer contains the input image or sequence of images. In this layer, the number of neurons equals the number of features (which is the number of pixels in image data).

Convolutional Layer(s) – this layer applies filters or 'kernels' to your image. These filters are smaller than your image size. A filter crosses your image in 'strides', and after each stride the model returns a value denoting how closely the portion of the image matches the filter. What are Convolutional Neural Networks (CNN)? is a short video that helps illustrate this concept.

Pooling Layer(s) – this layer's job is to reduce the size, which makes the model faster and combats overfitting. This layer summarizes each filter-sized portion of the image. In other words, for a group of features or pixels, the pooling layer returns a single feature that represents the group. Two common approaches to pooling are (1) average pooling, where the layer returns an average of the features and (2) max pooling, where the layer returns the maximum-valued feature. Geeks for Geeks goes into further detail in CNN | Introduction to Pooling Layer.

Fully Connected Layer(s) – this layer computes the final task.

Output Layer – this layer takes the final calculation and converts it to a probability score for each class.

Building an Architecture in PyTorch

Now that you understand what each of the layers do, let's look into how to put them together to build a CNN. During the hackathon if you would like to refine the provided CNN model's architecture to improve your f1-score, it will be important to understand how the model is built.

After loading and preprocessing your data , it's time to build a model. Start by instantiating the CNN_Model class and feeding it a Module object. Inside, you must define two functions:

- **init** – where the layers are defined
- **forward** – where the layers are put together

If you are looking for a great video, Convolutional Neural Nets Explained and Implemented in Python (PyTorch) is one of my favorite resources on the topic. If you prefer a text version, there is also Visual Guide to Applied Convolution Neural Networks.

Let's look at my other favorite resource to get a toy example to work through together (Convolutional Neural Network (CNN) – PyTorch Begginner14).

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # -> n, 3, 32, 32
        x = self.pool(F.relu(self.conv1(x)))  # -> n, 6, 14, 14
        x = self.pool(F.relu(self.conv2(x)))  # -> n, 16, 5, 5
        x = x.view(-1, 16 * 5 * 5)            # -> n, 400
        x = F.relu(self.fc1(x))               # -> n, 120
        x = F.relu(self.fc2(x))               # -> n, 84
        x = self.fc3(x)                       # -> n, 10
        return x
```

As you can see, within **init** you can create a convolutional layer, a pooling layer, a fully connected layer, and so much more! Some notes on creating the layers:

- `Conv2d` – our convolutional layers
    - Input: input channels, output channels, and kernel size.
        - In `conv1`, we input 3 channels because we are dealing with images that have 3 color channels (RGB – red, green, blue). We output 6 channels, and we create a kernel that is 5x5 pixels.
        - In `conv2`, we input 6 channels because we must take the output of the conv1 (the last convolutional layer) as input. We then output 16 channels and use a kernel that is 5x5 pixels.
    - Official documentation can be found on CONV2D.
- `MadxPool2d` – our pooling layer
    - Input: kernel size, stride
        - In `pool,` we create a pooling layer that has a size of 2x2 pixels and a stride of 2.
    - Official documentation can be found on MAXPOOL2D.
- `Linear` – our fully connected layers
    - Input: input features, output features
        - In `fc1`, the input features may look slightly strange. Let's look into why. The input into this first fully connected layer depends on the size of the last image produced before flattening the images into linear values. This last image has 16 channels and is 5x5 pixels (see explanation starting at the end of the next page). So, our input feature size is 16*5*5. The output feature size is more arbitrary, and we select 120.
        - In `fc2`, the input feature size is simply the output feature sizeof `fc2`. Again the output feature size is fairly arbitrary, and we select 84.

    - In `fc3`, the input feature size is again the output feature size of the last fully connected layer. The final output feature size has to match the number of classes, which for us is 10.
  - Official documentation can be found on LINEAR.

After creating all of our layers, we can put them together in our **forward** function. Here are a few notes on the code above:

- The `n` in the comments is your batch size. The 4-element comments are composed of: batch size, channel size, image size. The 2-element comments are composed of: batch size, output feature size.
- In our first line, we start out with our first convolutional layer, passed to an activation function, passed to a pooling layer. Then, we do the same thing but instead with our second convolutional layer.
  - If you are not familiar with activation functions, Activation Functions – PyTorch Beginner 12 has a great, short video explaining how they work.
- Next, we need to flatten our images so that they can go through the fully connected layers. For this, we use the `view()` function. We pass -1 as the first argument, which will make PyTorch automatically define the argument. We pass the most recent image size as the second argument.
- Then, we call our first two fully connected layers and pass both of them to an activation function. Last but not least, we call our final fully connected layer to get our output results.

Do you want to understand the size of your convolution output? Let's dig into it! The size is given by this equation:

$$size = \lfloor \frac{n + 2(padding) - kernel}{stride} \rfloor + 1$$

where:

n = image size (starting at 32)
padding = 0
kernel size = 5
stride = 1

Let's look at it-

`conv1`: The first layer is a convolutional layer. And we pass an image with 3 channels and of size 32x32:

Code	Equation
`self.conv1 = nn.Conv2d(3, 6, 5)`	$\frac{32 + 2(0) - 5}{1} + 1 = 28$

Output image size: ([6, 28, 28]). Note that the first element denotes new the number of channels, and the last two elements denote the new image size. So, our output is an image with 6 channels and a size of 28x28. This information can also be obtained by calling `print(x.shape)` after running each layer.

`pool:` Next is our pooling layer. Our kernel size is 2x2, and our stride is 2, so we will reduce our image size by a factor of 2:

Code	Equation
`self.pool = nn.MaxPool2d(2, 2)`	$\frac{28}{2} = 14$

Output image size: ([6, 14, 14]). Note that the channel size does not change because channel size is not altered by the pooling function.

`conv2:` Then we pass this newly-sized image to our second convolutional layer:

Code	Equation
`self.conv2 = nn.Conv2d(6, 16, 5)`	$\frac{14 + 2(0) - 5}{1} + 1 = 10$

Output image size: ([16, 10, 10]).

`pool:` Next, we have another pooling layer. Let's calculate the size of this layer's output image:

Code	Equation
`self.pool = nn.MaxPool2d(2, 2)`	$\frac{10}{2} = 5$

Output image size: ([16, 5, 5]).

`fc1:` At last, we have our value calculated to pass as an input feature to the first fully connected layer!

Now, you can apply this to the hackathon model to understand its layers and output. If you decide to alter the model, you may want to use this information to evaluate how your changes affect the model.

Data Preprocessing

Downsampling and upsampling

Both of these techniques have to do with altering the pixel frequency in your image. As you might imagine, downsampling is the process of decreasing the frequency of pixels while upsampling is the process of increasing the frequency of pixels. Below are several useful resources to help guide you through the process:

- Spatial resolution (down sampling and up sampling) in image processing provides examples for how to perform both of these techniques in OpenCV.
- If instead you want to try *subsampling*, check out Image Subsampling and Downsampling. Subsampling is similar to downsampling in that it decreases the pixel frequency of an image. Downsampling accomplishes this by averaging a groups of pixels into single pixels. Alternatively, subsampling accomplishes this by simply discarding every other row and column. It is worthwhile to note that subsampling is faster, but it produces images that are less smooth.

Color channel setup using PyTorch

Note that this is not the method used in the competition workbook. If you want to use this method, you may need to delete the normalization code in the competition workbook and replace it with the method described here. Try it! See which method gives you a better f1-score.

If you want to use a PyTorch model, you have to setup your data to be input the way that the model is expecting it. This means that you will need to normalize your data in a specific way.

Pretrained PyTorch models expect data with RGB channels setup so that each channel has a precalculated value between 0 and 1 (as opposed to the typical 0 to 255). One way to do this is by using the normalize function and passing two precalculated lists to it:

- A list of mean inputs: [0.485, 0.456, 0.406]
  - Here, *mean* is a list of the mean of each color channel. In other words the numerical values can be mapped to [red pixel mean, blue pixel mean, green pixel mean]
- A list of std inputs: [0.229, 0.224, 0.225]
  - Here, *std* is a list of the standard deviation of each color channel. In other words the numerical values can be mapped to [red pixel standard deviation, blue pixel standard deviation, green pixel standard deviation]

Here is some example code from Building Custom Image Datasets in PyTorch:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

However, you will not be working with a pretrained PyTorch model, so it may or may not be better to try a different set of values in the mean and std lists. This approximately 1 minute portion of a video on convolutional neural networks details how to calculate a set of values for mean and std when using a non-pretrained PyTorch model.

Other preprocessing functions

Along with Normalizing your data, there are other things you can do to make your data more digestible. Conveniently, you can chain all your preprocessing together using torchvision.transforms.Compose. TORCHVISION.TRANSFORMS provides documentation and a great coding example:

```
>>> transforms.Compose([
>>>     transforms.CenterCrop(10),
>>>     transforms.ToTensor(),
>>> ])
```

Along with the CenterCrop and ToTensor functions, you can include a Normalization function and any other preprocessing functions you would like.

OpenCV Functions

What is OpenCV?

OpenCV is an open source library for C++, Python, and Java that can be used to process images or videos. In our case, we will use OpenCV to prepare a set of images for model ingestion. OpenCV – Overview has lots more detail if you're interested in delving deeper into this library.

Basic image operations

The documentation on basic image operations in OpenCV can be found here:

- Operations with Images
- Basic Operations on Images

You can see a demo of how to use these functions on an image of a C130 Hercules in your `open_cv_basics.ipynb` workbook. Google Collab is a great resource for opening the workbook.

Image augmentation operations

Some of OpenCV and PyTorch's official documentation on operations that can be used for image augmentation are below:

- Image Thresholding
- Canny Edge Detection
- Transforming and Augmenting Images

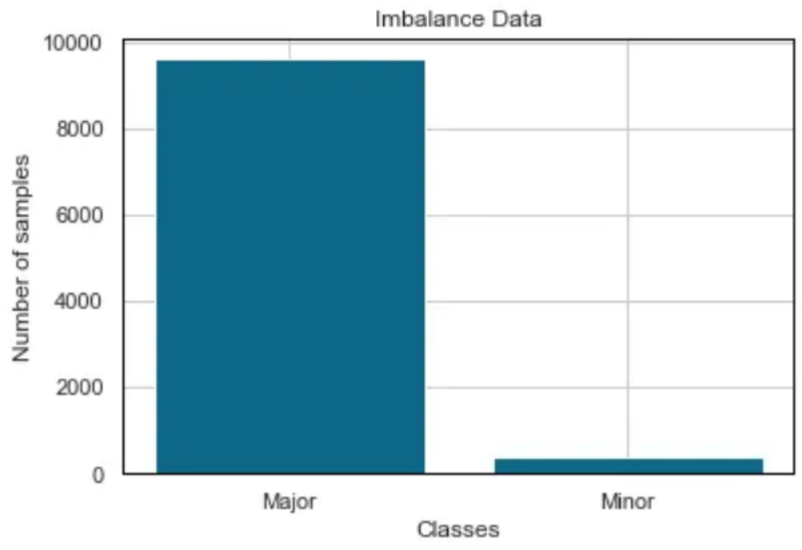
This additional documentation from OpenCV may be useful as well:

- Image Filtering
- Changing the contrast and brightness of an image!
- Color Space Conversions

Data Imbalance

What it is

An imbalanced dataset is one that contains majority and minority classes, meaning that one class has a lot more data entries than the other. Here's a visual:



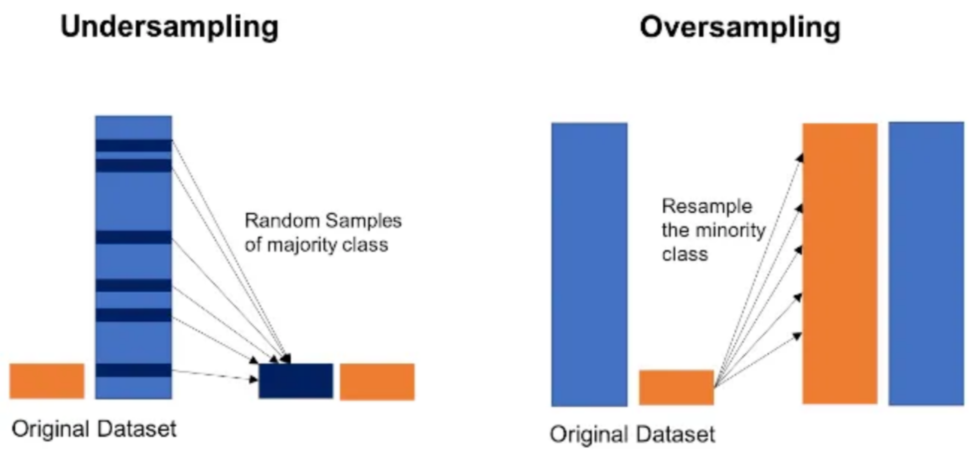
From How to Handle Imbalance Data and Small Training Sets in ML

Why does an imbalanced dataset matter? The biggest problem with this is that your model may become well-attuned to your majority class while not developing the ability to identify your minority class very well.

Strategies for handling it

There are quite a few approaches to choose from to combat imbalanced datasets:

1. Oversampling and undersampling
    a. Oversampling and undersampling are another set of techniques you may find useful to implement. Oversampling is the process of adding data entries to your class while undersampling is the process of removing them. It is common to use data augmentation to perform oversampling, which is covered in the next bullet. For undersampling, you simply have to remove entries. Be careful not to lose important information or create an inaccurate representation of your class in the process! Here is an graphic to illustrate the two methods:



From How to Deal with Imbalanced Data

        i. Geeks for Geeks' Introduction to Resampling Methods gives methods and examples of how to implement oversampling and undersampling.

2. Image augmentation
    a. One data imbalance strategy is to use image augmentation to perform oversampling on a minority class. So, what is image augmentation? Simply put, image augmentation is a set of techniques used to provide a wider base of training data for your CNN model. Just a few of these techniques include shifting your image horizontally or vertically, zooming in or out, altering brightness, etc. So whether you want to combat data imbalance or simply provide a larger set of training data to your model, this may be a great topic to look into!
        i. TRANSFORMING AND AUGMENTING IMAGES is the official PyTorch documentation, and it can guide you through this process.
        ii. A Survey on Image Data Augmentation for Deep Learning goes into deeper detail about state-of-the-art image augmentation techniques.
        iii. Data augmentation for imbalanced blood cell image classification is a research paper that demonstrates how to use data augmentation to combat data imbalance (by performing oversampling on a minority class).

3. Including weights for each class in a loss function
    a. If there is a data imbalance, one way to combat it is to treat the loss function differently for the majority and minority classes. If you treat them all the same, then the model may end up being biased toward the majority class. If however, you treat misclassifications on a minority class as more important, then the model will not be as likely to train with a bias towards the majority class.
        i. The fourth point in 4 Ways to Improve Class Imbalance illustrates the concept of including weights for each class in a loss function.
        ii. Class weights for categorical loss explains the mathematics behind this concept.
        iii. The second point in How to Handle Imbalance Data and Small Training Sets in ML gives a code example for how to calculate and implement your weighted loss function.

4. If you are interested in other ways to combat data imbalance, check out one of the following resources:
    a. 4 Ways to Improve Class Imbalance for Image Data
    b. How to Handle Imbalance Data and Small Training Sets in ML

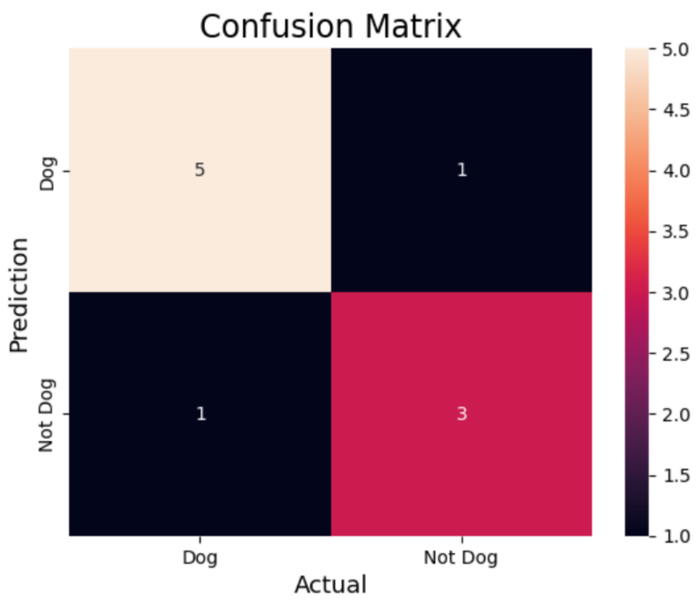
Metrics

After we build the model, we should determine how accurate it is. How are we supposed to do this? Well, there are *many* approaches, but two important ones are the ***confusion matrix*** and ***f1-score***.

Confusion matrix

A confusion matrix is a metric used on classification algorithms. In this matrix, each class is given a row and a column. The column class is the actual class an item belongs to, while the row class is the model's predicted class. The more values that have matching column and row values, the more accurate your model is.

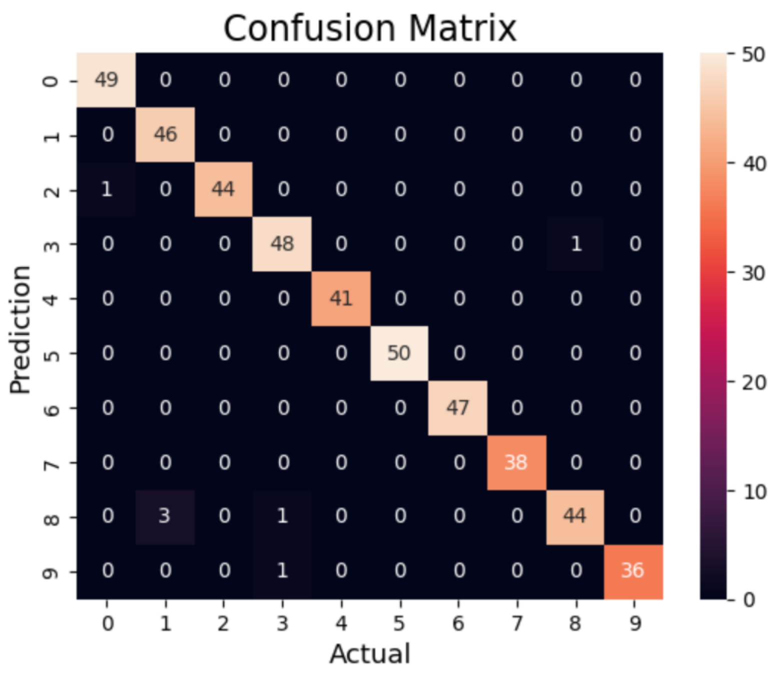
To start off, let's look at a binary classification algorithm. Below, you can see a confusion matrix for a model that predicts whether an image belong to class "Dog" or to class "Not Dog." As you can see, the model correctly predicts five dog images as dog images and only incorrectly predicts one non-dog image as a dog image. For the "Not Dog" class, the model correctly predicts 3 non-dog images and incorrectly predicts 1 dog image as a non-dog image.



From Confusion Matrix in Machine Learning

So, with only two classes, the downward diagonal contains the correctly predicted items while the upward diagonal contains the incorrect predictions. Now, how do we read or write a confusion matrix for more than two classes?

A confusion matrix for a multi-class confusion matrix is quite similar. The more values that have matching column and row values, the more accurate your model is. Let's look at an example of a classification model that predicts which number a handwritten digit is:



Prediction \ Actual	0	1	2	3	4	5	6	7	8	9
0	49	0	0	0	0	0	0	0	0	0
1	0	46	0	0	0	0	0	0	0	0
2	1	0	44	0	0	0	0	0	0	0
3	0	0	0	48	0	0	0	0	1	0
4	0	0	0	0	41	0	0	0	0	0
5	0	0	0	0	0	50	0	0	0	0
6	0	0	0	0	0	0	47	0	0	0
7	0	0	0	0	0	0	0	38	0	0
8	0	3	0	1	0	0	0	0	44	0
9	0	0	0	1	0	0	0	0	0	36

From Confusion Matrix in Machine Learning

As you can see, the downward diagonal still contains all the accurate predictions. However, it is important to note that instead of *one* possible incorrect prediction, there are now *nine*. So for example, where column 1 crosses row 8, there are three items that belong to class 1 that were classified as members of class 8. Also, where column 8 crosses row 3, there is 1 item that belongs to class 8 that was classified as a member of class 3.

F1-score

In order to understand f1-score, you must first become familiar with ***precision*** and ***recall***.

Precision is obtained by calculating the percentage of accurate predictions across all items classified by the model as belonging to the same class. You can see the calculation here:

$$precision = \frac{true\ positive\ predicted\ as\ true\ positive}{results\ predicted\ as\ true\ positive} = \frac{TP}{TP + FP}$$

So, given our dog example, the precision of the model on the “Dog” class is:

$$precision = \frac{5}{5+1} = \frac{5}{6} = 0.83$$

On the other hand, the precision of the model on the “Not Dog” class is:

$$precision = \frac{3}{3+1} = \frac{3}{4} = 0.75$$

When applied to a multi-class model, the only difference is that you must consider the additional classes. For example, the precision of the hand-writing classification model on the number 3 is:

$$precision = \frac{48}{48+1+1} = \frac{48}{50} = 0.96$$

Recall is similar, but not quite. It is obtained by calculating the percentage of accurate predictions across all items belonging to the same class. You can see the calculation here:

$$recall = \frac{true\ positive\ predicted\ as\ true\ positive}{all\ actually\ positive\ values} = \frac{TP}{TP+FN}$$

So, while precision is calculated across a column, recall is calculated across a row. For both values, the closer you get to 1, the better.

Now, f1-score is a combination of the methods to account for the information provided by both:

$$f1 - score = \left(\frac{recall^{-1} + precision^{-1}}{2}\right)^{-1} = 2 * \frac{precision\ \cdot recall}{precision + recall}$$

Similar to precision and recall, the closer your f1-score is to 1, the more accurate your model is.

If interested in further research into metrics, here are some great resources:

- Confusion matrix and f1-score
- Multi-Class Metrics Made Simple, Part I: Precision and Recall

Back to top.