# Quantitative Big Imaging - Building and Augmenting Datasets

**Anders Kaestner**

**Mar 31, 2022**

# CONTENTS

This is the lecture notes for the 2nd lecture of the Quantitative big imaging class given during the spring semester 2022 at ETH Zurich, Switzerland.

CHAPTER

ONE

# GROUND TRUTH: BUILDING AND AUGMENTING DATASETS

**Quantitative Big Imaging** ETHZ: 227-0966-00L

## 1.1 Today's lecture

**Creating Datasets**

- Famous Datasets
- Types of Datasets
- What makes a good dataet?
- Building your own
- "scrape, mine, move, annotate, review, and preprocess" - Kathy Scott
- tools to use
- simulation

**Augmentation**

- How can you artifically increase the size of your dataset?
- What are the limits of these increases

**Baselines**

- What is a baseline?
- Example: Nearest Neighbor

## 1.2 Let's load some modules for the notebook

```
import matplotlib.pyplot as plt
import matplotlib    as mpl
import numpy         as np
import skimage       as ski
import skimage.io    as io
from skimage.morphology import disk
import scipy.ndimage as ndimage
from keras.datasets  import mnist
from skimage.util    import montage as montage2d
```

(continues on next page)

(continued from previous page)

```
%matplotlib inline
mpl.rcParams['figure.dpi'] = 100
```

## 1.3 References

- Revisiting **Unreasonable Effectiveness of Data** in Deep Learning Era
- Data science … without any data
- Building Datasets
  - Python Machine Learning 2nd Edition by Sebastian Raschka, Packt Publishing Ltd. 2017
  - Chapter 2: Building Good Datasets:
  - A Standardised Approach for Preparing Imaging Data for Machine Learning Tasks in Radiology
- Creating Datasets / Crowdsourcing
- Mindcontrol: A web application for brain segmentation quality control
- Combining citizen science and deep learning to amplify expertise in neuroimaging
- Augmentation tools
- ImgAug
- Augmentor

## 1.4 Motivation

**Why other peoples data?**

Most of you taking this class are rightfully excited to learn about new tools and algorithms to analyzing *your* data.

This lecture is a bit of an anomaly and perhaps disappointment because it doesn't cover any algorithms, or tools.

- You might ask, why are we spending so much time on datasets?
- You already collected data (sometimes lots of it) that is why you took this class?!

… let's see what some other people say

### 1.4.1 Sean Taylor (Research Scientist at Facebook)

This tweet tells us that you shouldn't put too much belief in AI without providing carefully prepared data set. Machine learning methods perform only so good as the data it was trained with. You need a data set that covers all extremes of the fenomena that you want to model.

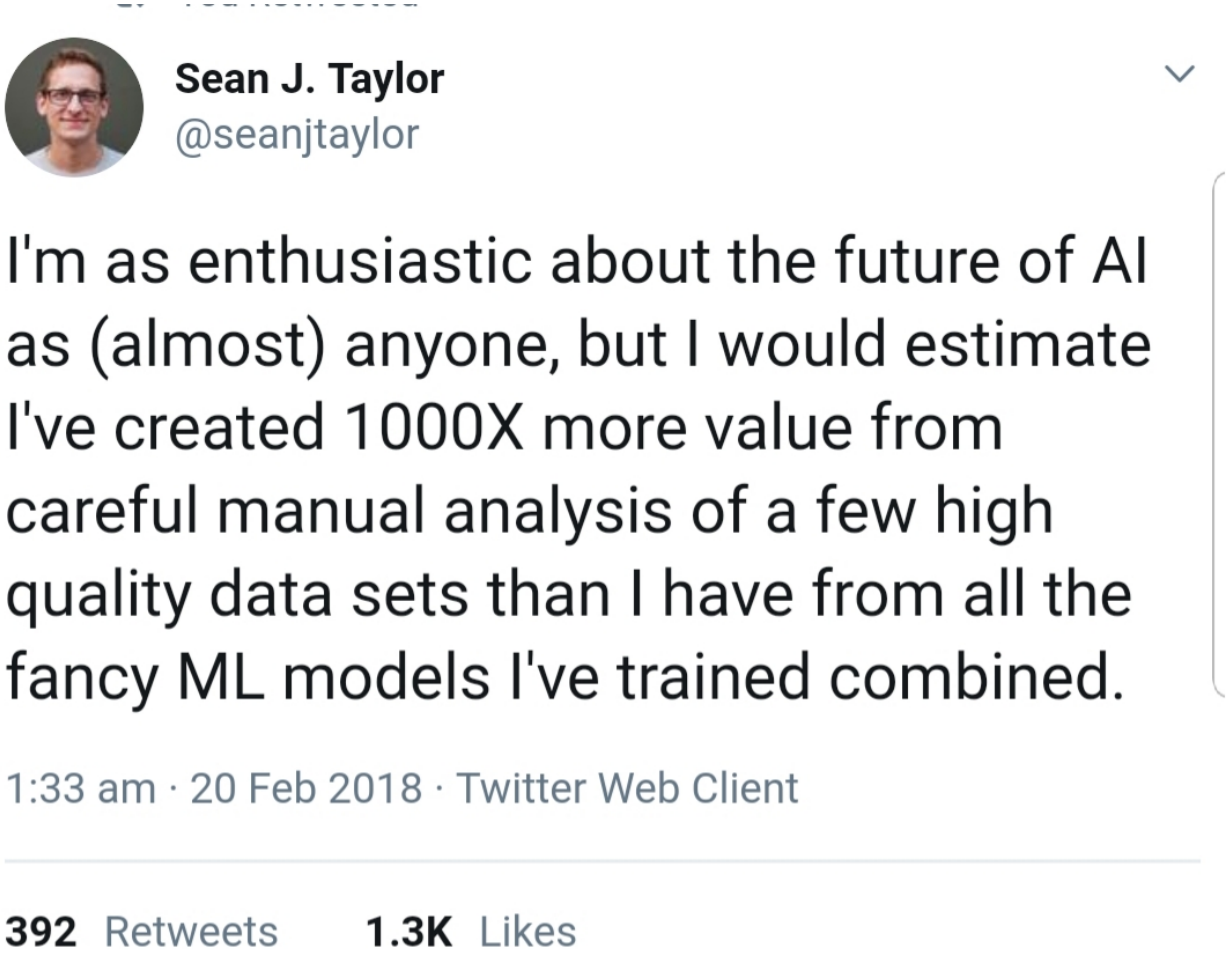



Fig. 1.1: Realistic thoughts about AI.

### 1.4.2 Andrej Karpathy (Director of AI at Tesla)

This slide by Andrej Karpathy shows the importance of correct data set in a machine learning project. Typically, you should spend much more time on collecting representative data for your models than building the models. Unfortunately, this is not the case for many PhD projects where the data usually is scarse. Much for the reason that it is really hard to come by the data. You may only have a few beam slots allocated for your experiments and this is the data you have to live with.

### 1.4.3 Kathy Scott (Image Analytics Lead at Planet Labs)

Yet another tweet that implies that many data scientist actually spend more time on preparing the data than developing new models. The training is less labor demanding, the computer is doing that part of the job.

## 1.5 Data is important

It probably isn't the *new* oil, but it forms an essential component for building modern tools today.

Testing good algorithms *requires* good data

- If you don't know what to expect – *how do you know your algorithm worked*?
- If you have dozens of edge cases – *how can you make sure it works on each one*?
- If a new algorithm is developed every few hours – *how can you be confident they actually work better*?
  - facebook's site has a new version multiple times per day and their app every other day

For machine learning, even building algorithms requires good data

- If you count cells maybe you can write your own algorithm,
- but if you are trying to detect *subtle changes* in cell structure that indicate cancer you probably can't write a list of simple mathematical rules yourself.

## 1.6 Data is reusable

- Well organized and structured data is very easy to reuse.
- Another project can easily combine your data with their data in order to get even better results.

Algorithms are often often only prototypes

- messy,
- complicated,
- poorly written,

… especially so if written by students trying to graduate on time.

**Data recycling saves time and improves performance**

Why you need to improve your training data, and how to do it

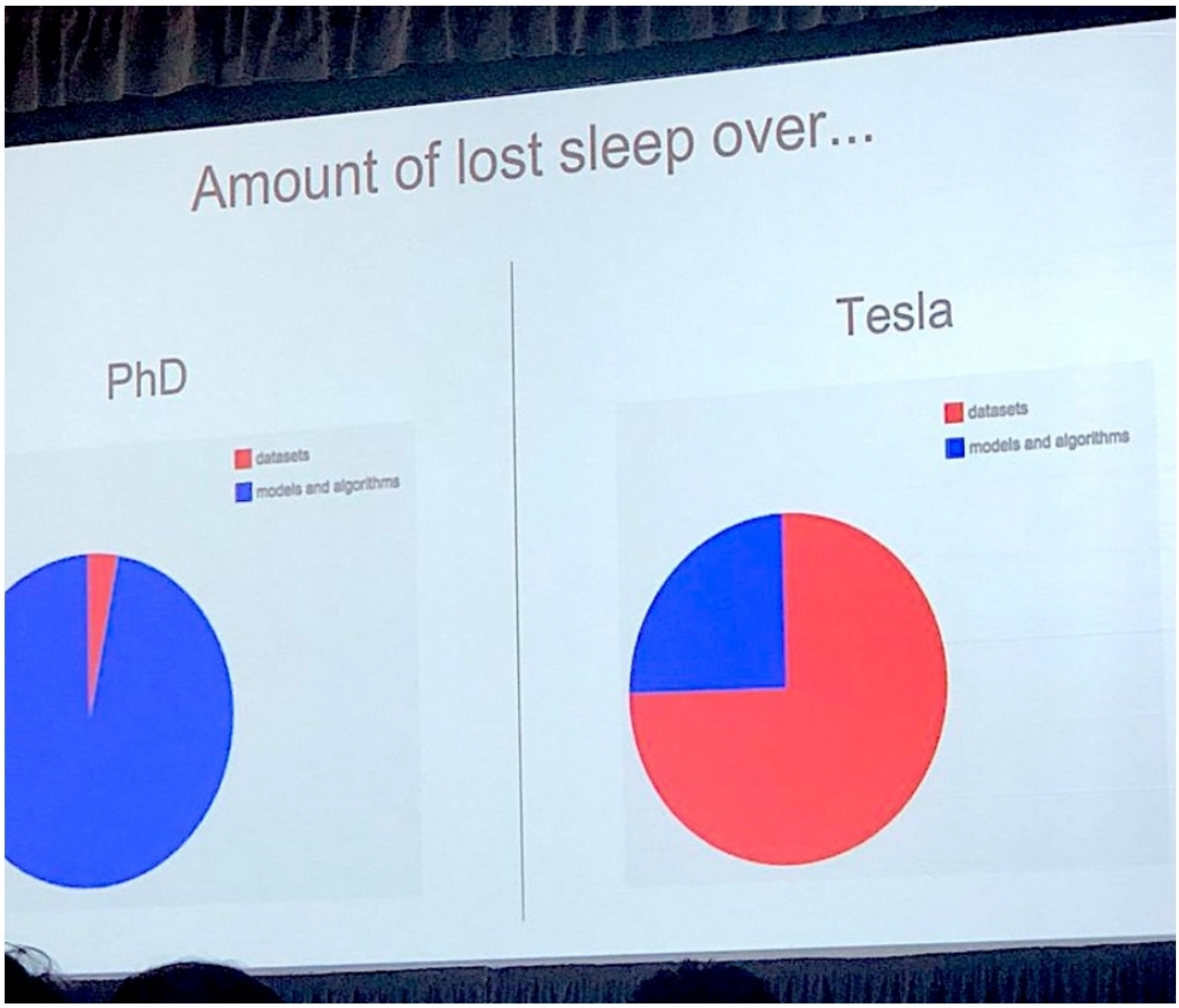



*Photo by Lisha Li*

Andrej Karpathy showed this slide as part of his talk at Train AI and I loved it! It captures the difference between deep learning

Fig. 1.2: Time spent on different tasks.

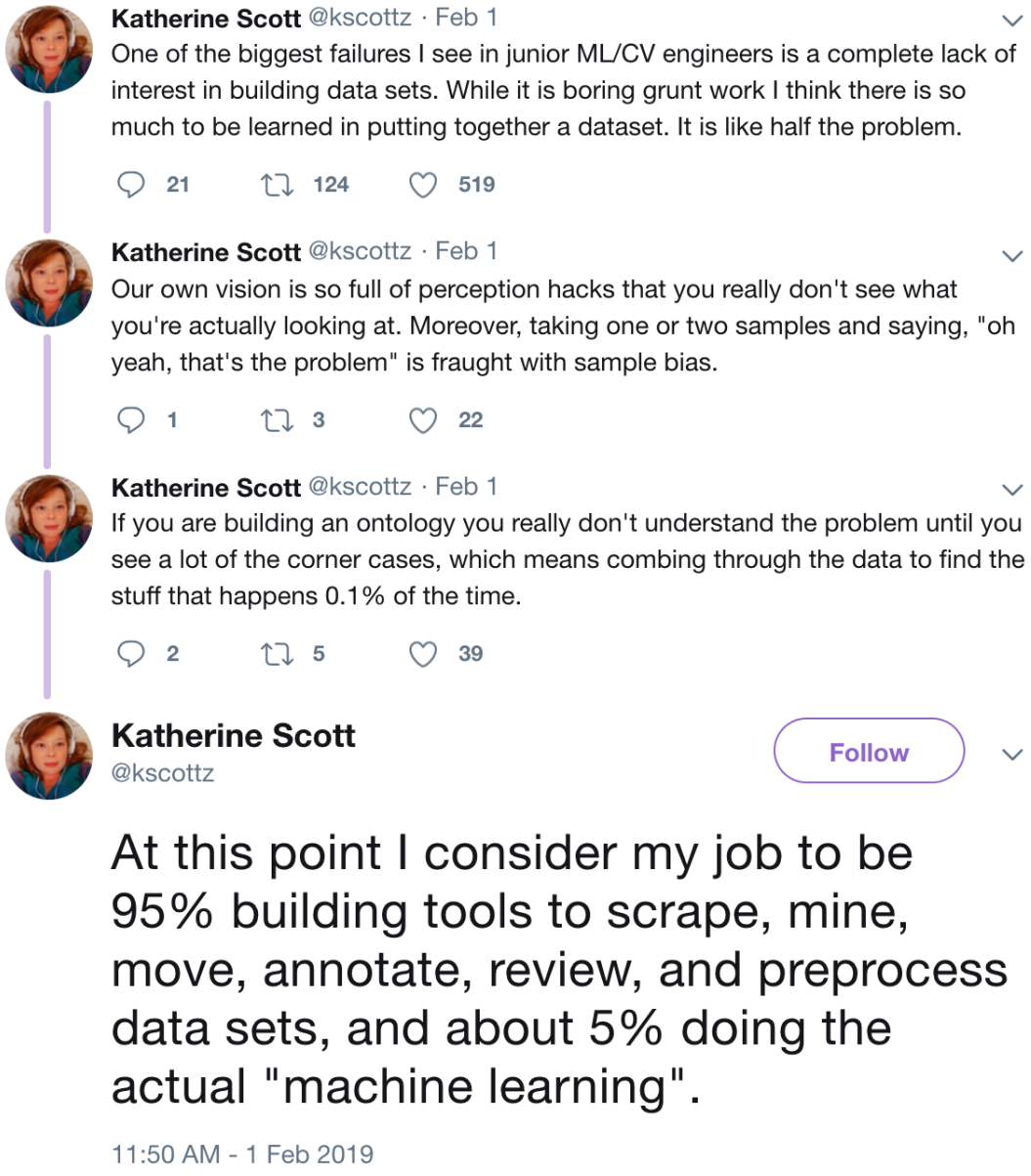



Fig. 1.3: The importance to spend sufficient time on data preparation.

CHAPTER

TWO

# FAMOUS DATASETS

The primary success of datasets has been shown through the most famous datasets collected.

Here I show

- Two of the most famous general datasets
  - MNIST Digits
  - ImageNET
- and one of the most famous medical datasets.
  - BRATS

The famous datasets are important for basic network training.

## 2.1 MNIST Digits

Modified NIST (National Institute of Standards and Technology) created a list of handwritten digits.

This list is a popular starting point for many machine learning projects. The images are already labeled and are also nicely prepared to about the same size and also very high SNR. These properties makes it a great toy data set for first testing.

## 2.2 ImageNet

- ImageNet is an image database
  - organized according to the WordNet hierarchy (currently only the nouns),
  - each node of the hierarchy is depicted by hundreds and thousands of images.
- 1000 different categories and >1M images.
- Not just dog/cat, but wolf vs german shepard,

CNN architectures

Fig. 2.1: A selection of hand written numbers from the MNIST data base

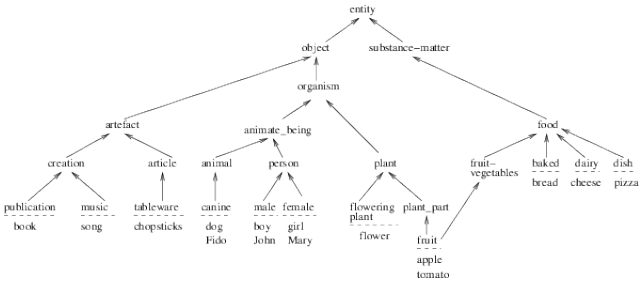



Fig. 2.2: Hierarchial structure of the WordNet database.

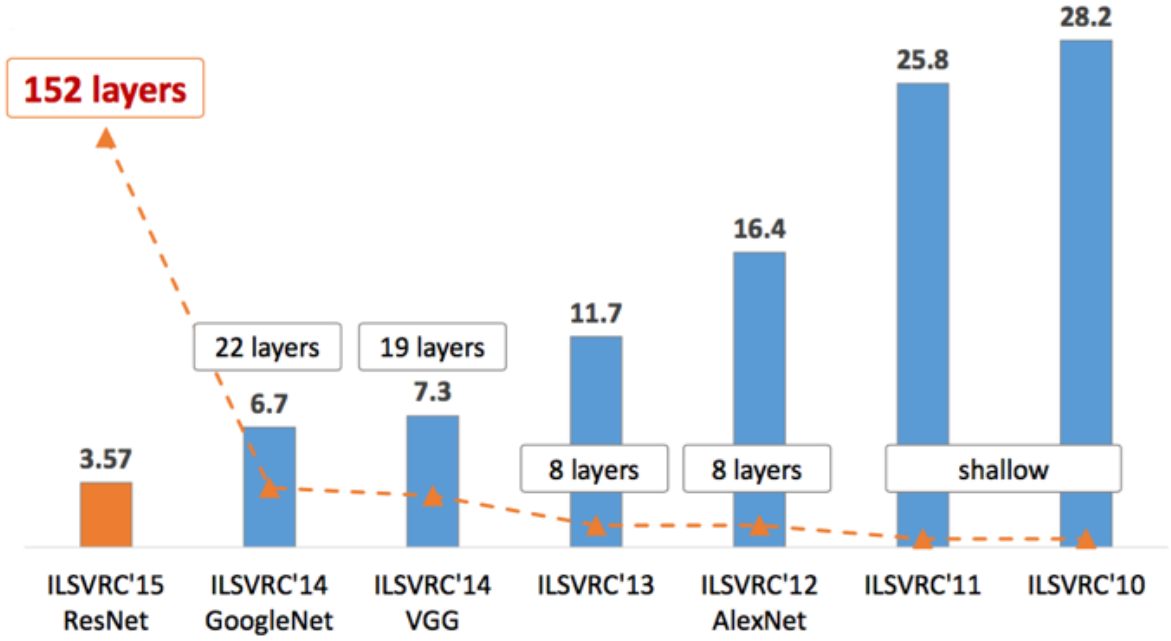



Fig. 2.3: Error rates for different classifiers on the same data set.

## 2.3 BRATS

Segmenting Tumors in Multimodal MRI Brain Images.

## 2.4 What story did these datasets tell?

Each of these datasets is very different from images with fewer than 1000 pixels to images with more than 100MPx, but what they have in common is how their analysis has changed.

All of these datasets used to be analyzed by domain experts with hand-crafted features.

- A handwriting expert using graph topology to assign images to digits
- A computer vision expert using gradients common in faces to identify people in ImageNet
- A biomedical engineer using knowledge of different modalities to fuse them together and cluster healthy and tumorous tissue

Starting in the early 2010s, the approaches of deep learning began to improve and become more computationally efficient. With these techniques groups with **absolutely no domain knowledge** could begin building algorithms and winning contests based on these datasets.

All of these datasets used to be analyzed by **domain experts** with hand-crafted features.

- A handwriting expert using graph topology
- A computer vision expert to identify people in ImageNet
- A biomedical engineer cluster healthy and tumorous tissue
- the approaches of deep learning began to improve and become more computationally efficient.

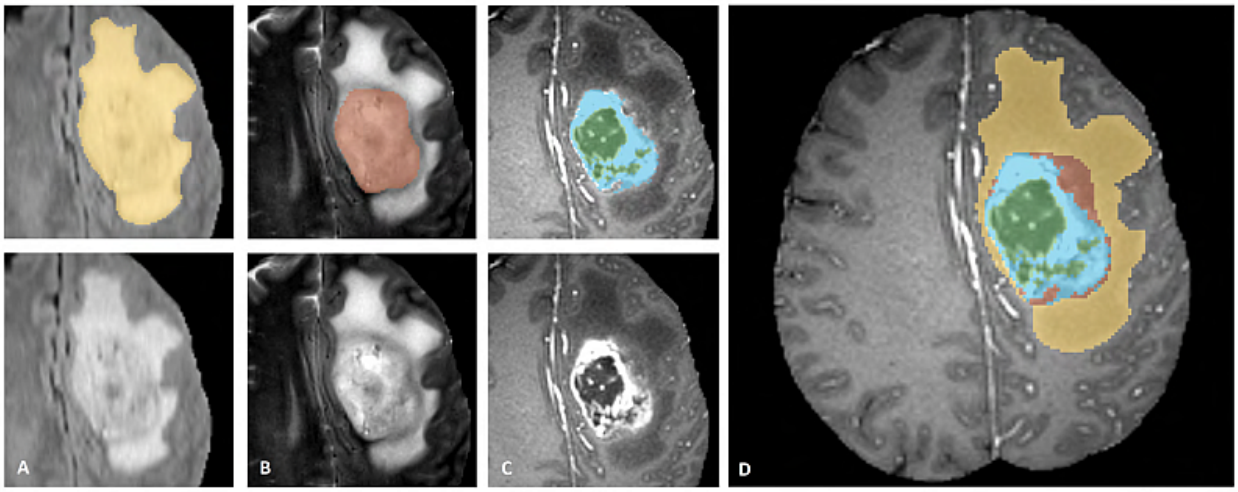



Fig. 2.4: Images of brain tumors from the BRATS database.

Fig. 2.5: Domain experts use their experience to analyze data

Fig. 2.6: Data scientist don't have domain specific knowledge, they use available data for the analysis.

- groups with **absolutely no domain knowledge** could winning contests based on datasets

## 2.5 So Deep Learning always wins?

No, that isn't the point of this lecture.

Even if you aren't using deep learning the point of these stories is having

- well-labeled,
- structured,
- and organized datasets

makes your problem *a lot more accessible* for other groups and enables a variety of different approaches to be tried.

Ultimately it enables better solutions to be made and you to be confident that the solutions are in fact better.

### 2.5.1 How to work with someone else's data

> Inherited datasets are like inherited toothbrushes: using them is an act of desperation.
>
> Collecting your own data is a luxury not everyone can afford.
>
> Inherited data are easier to get but harder to trust.

C. Kozyrkov, 2020

### 2.5.2 The FAIR principle

Open data is a central requirement these days.

- __F__indable
- __A__ccessible
- __I__nteroperable
- __R__eusable

Wilkinson et al. 2016

#### PaNOSC

The Photon and Neutron Open Science Cloud (PaNOSC)

## 2.6 Other Datasets

- Grand-Challenge.org a large number of challenges in the biomedical area
- Kaggle Datasets
- Google Dataset Search
- Wikipedia provides a comprehensive list categorized into different topics

## 2.7 What makes a good dataset?

A good data set is characterized by

- Large amount
- Diversity
- Annotations

This means that…

### 2.7.1 Lots of images

- Small datasets can be useful, but here the bigger the better
- Particularly if you have:
  - Complicated problems
  - Very subtle differences (ie a lung tumor looks mostly like normal lung tissue but it is in a place it shouldn't be)
  - Class imbalance

### 2.7.2 Lots of diversity

- Is it what data 'in the wild' really looks like?
- Lots of different
  - Scanners/reconstruction algorithms,
  - noise levels,
  - illumination types,
  - rotation,
  - colors, …
- Many examples from different categories
  - *if you only have one male with breast cancer it will be hard to generalize exactly what that looks like*

### 2.7.3 Meaningful labels

- Clear task or question
- Unambiguous (would multiple different labelers come to the same conclusion)
- Able to be derived from the image alone
  - *A label that someone cannot afford insurance is interesting but it would be nearly impossible to determine that from an X-ray of their lungs*
- Quantiative!
- Non-obvious
  - *A label saying an image is bright is not a helpful label because you could look at the histogram and say that*

CHAPTER

THREE

# PURPOSE OF DIFFERENT TYPES OF DATASETS

- Classification
- Regression
- Segmentation
- Detection
- Other

## 3.1 Classification

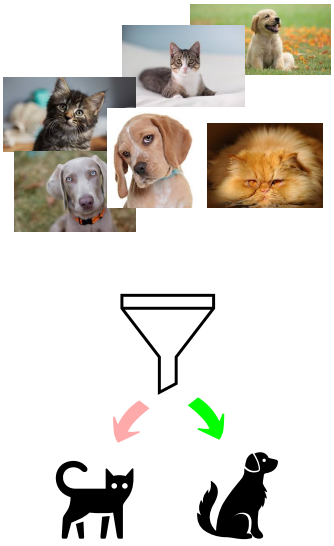


Fig. 3.1: Classification example with cats and dogs.

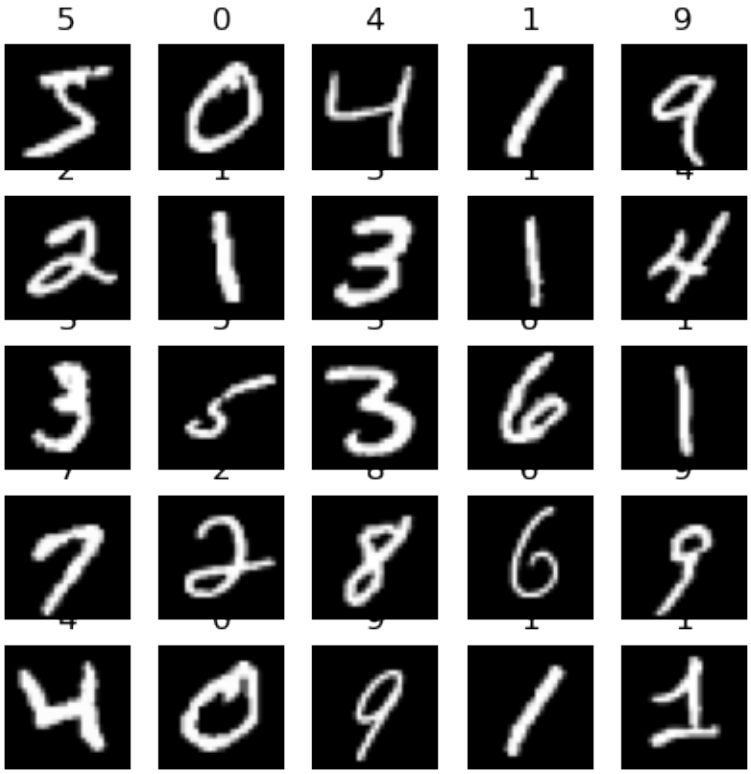
- Taking an image and putting it into a category
- Each image should have exactly one category
- The categories should be non-ordered
- Example:
- Cat vs Dog

- Cancer vs Healthy

### 3.1.1 Classification example

In classification you want to observe an image an quickly provide it with a category. Like in the MNIST example which is designed for recognition of handwritten numbers. Each image has a label telling which number it represents.

```
(img, label), _ = mnist.load_data()
fig, m_axs = plt.subplots(5, 5, figsize=(5, 5))
for c_ax, c_img, c_label in zip(m_axs.flatten(), img, label):
    c_ax.imshow(c_img, cmap='gray')
    c_ax.set_title(c_label)
    c_ax.axis('off')
```




## 3.2 Regression

Regression almost looks like classification at first sight. You still want to put a number related to the image content. But here it is not strictly bound to the provided categories but rather estimating a value which can be found on the regression line fitted to the data.

Taking an image and predicting one (or more) decimal values

- Examples:
- Value of a house from the picture taken by owner
- Risk of hurricane from satellite image

### 3.2.1 Regression example Age from X-Rays

This dataset contains a collection of X-ray radiographs of hands. The purpose of the data is to estimate the age of a child based on the radiograph. This can be done using a regression model.

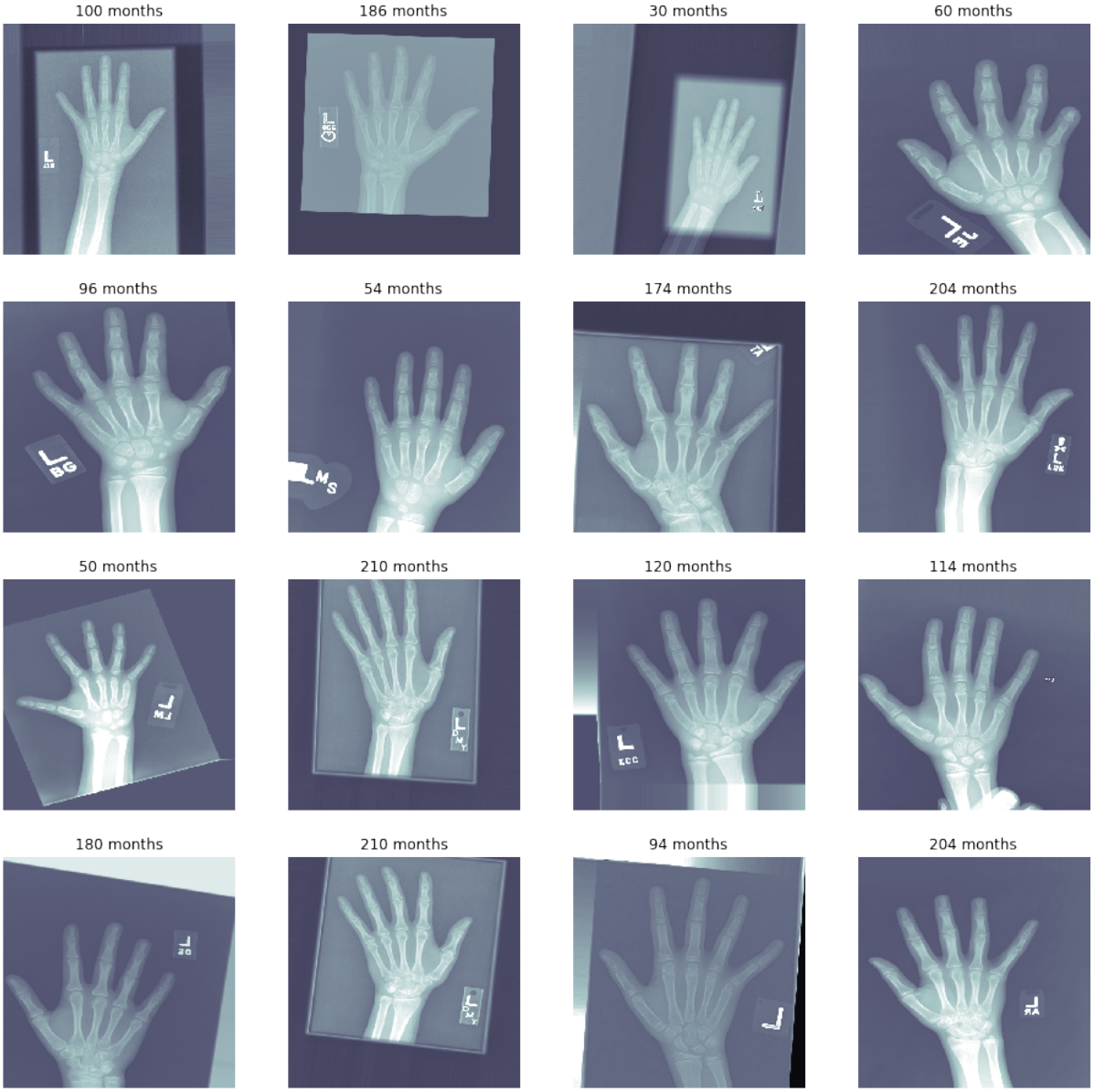



Fig. 3.2: A collection of X-ray images from children at different ages.

More details

## 3.3 Segmentation

- Taking an image and predicting one (or more) values for each pixel
- Every pixel needs a label (and a pixel cannot have multiple labels)
- Typically limited to a few (less than 20) different types of objects

**Examples:**

- Where a tumor is from an image of the lungs?
- Where streets are from satellite images of a neighborhood?
- Where are the cats and dogs?

### 3.3.1 Segmentation example: Nuclei in Microscope Images

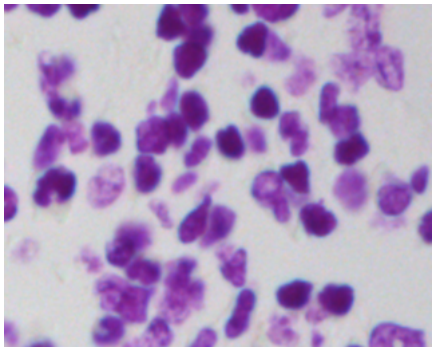


Fig. 3.3: Sample with cells.

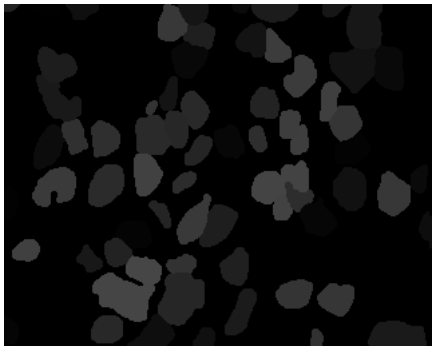


Fig. 3.4: Labelled cells.

More details on Kaggle

## 3.4 Detection

Detection is a combination of segmenation and classification in the sense that the location and extents of a feature is determined and is also categorized into some class. The extents don't have to be very precise, it is often a bounding box or a convex hull. This coarseness is sufficient for many applications.

- Taking an image and predicting where and which type of objects appear
- Generally bounding box rather than specific pixels
- Multiple objects can overlap

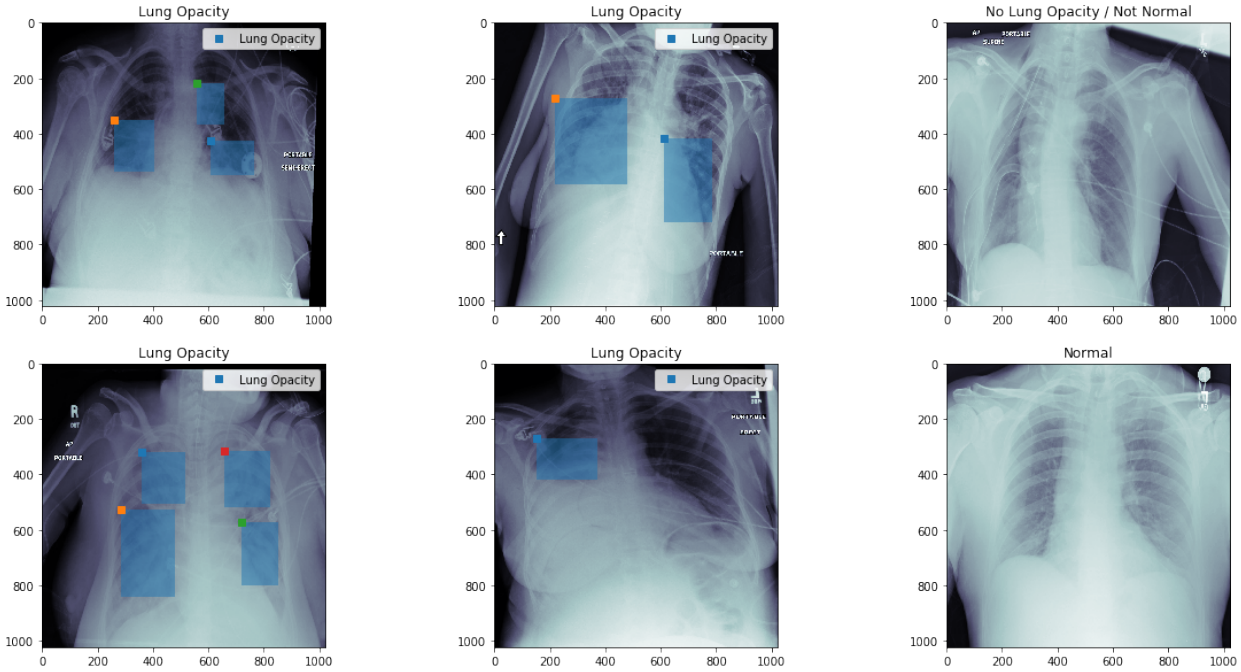



Fig. 3.5: Radiographs to detect opaque regions in X-Rays

### 3.4.1 Detection example: Opaque Regions in X-Rays

In this example the task is to detect opaque regions in lung X-ray images to provide a first indication for the physician who should make a diagnosis from the images. The used algorithm marks rectangles on region that are too opaque to be healthy.

More details on Kaggle

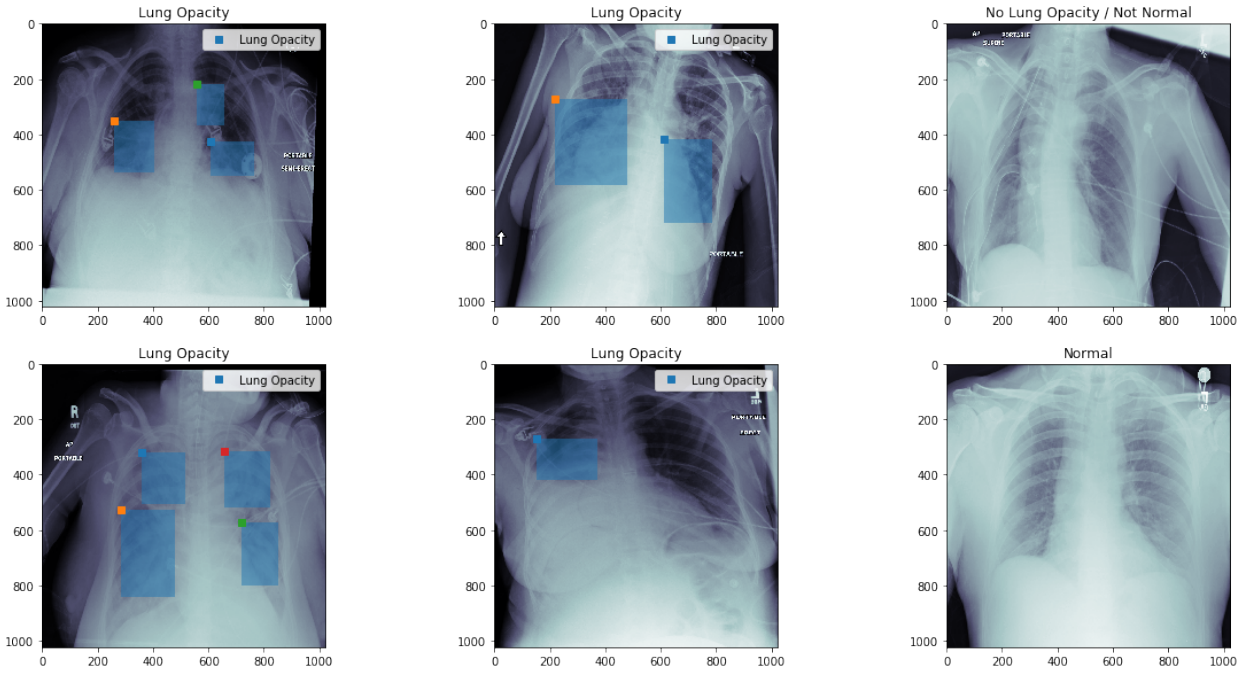



Fig. 3.6: Critical regions detected in lung radiographs.

## 3.5 Other

- Unlimited possibilities here
- Horses to Zebras

### 3.5.1 Image Enhancement

- Denoising Learning to See in the Dark
- Super-resolution

CHAPTER

FOUR

# BUILDING YOUR OWN DATA SETS

Finally, we arrive at your data! As you already have seen, it is a time consuming and labor intense task to collect and prepare data.

- Very time consuming
- Not a lot of great tools
- Very problem specific

It is however important to have well-organized data for the analysis.

## 4.1 Code-free

### 4.1.1 Classification

- Organize images into folders

### 4.1.2 Regression

- Create an excel file (first column image name, next columns to regress)

### 4.1.3 Segmentation / Object Detection

- Take FIJI or any paint application and manually draw region to be identified and save it as a grayscale image

## 4.2 Software for data labelling

### 4.2.1 Free tools

- Classification / Segmentation
- Classification/ Object Detection
- Classification (demo)
- Classification/ Detection
- Classification (Tinder for Brain MRI)

### 4.2.2 Commercial Approaches

- https://www.figure-eight.com/
- MightyAI / Spare5: https://mighty.ai/ https://app.spare5.com/fives/sign_in

### 4.2.3 Example: annotation of spots

Spots are outliers in radiography and very annoying when the images are used for tomography.

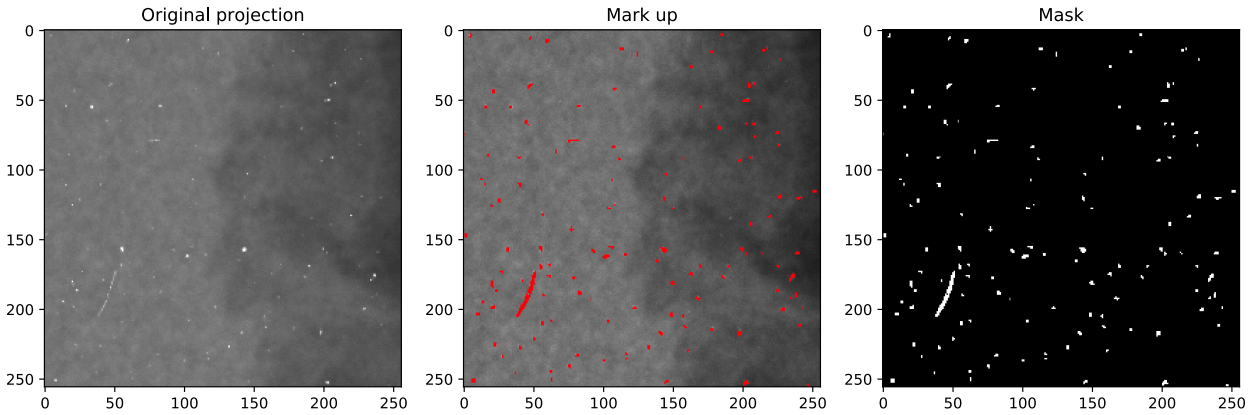



Fig. 4.1: Annotation of spot in a neutron radiograph.

- Image size 2048x2048
- Tools
  - Bitmap painting application
  - Drawing tablet
- Time to markup *8h*

## 4.3 Simulations

A further way to increase training data is to build a model of the features you want to train on. This approach has the advantage that you know where to look for the features which means the tedious annotation task is reduced to a minimum. The work rather lies in building a relieable model that should reflect the characteristics of features you want to segments. Once a valid model is built, it is easy to generate masses of data under variuos conditions.

Simulations can be done using:

- Geometric models
- Template models
- Physical models

Both augmented and simulated data should be combined with real data.

### 4.3.1 Simulation examples

Another way to enhance or expand your dataset is to use simulations

- already incorporate realistic data (game engines, 3D rendering, physics models)
- 100% accurate ground truth (original models)
- unlimited, risk-free playability (driving cars in the world is more dangerous)

#### Examples

- P. Fuchs et al. - Generating Meaningful Synthetic Ground Truth for Pore Detection in Cast Aluminum Parts, iCT 2019, Padova
- Playing for Data: Ground Truth from Computer Games
- Self-driving car
- Learning from simulated data

CHAPTER

FIVE

# DATASET PROBLEMS

Some of the issues which can come up with datasets are

- imbalance
- too few examples
- too homogenous
- and other possible problems

These lead to problems with the algorithms built on top of them.

## 5.1 Bias

Working with single sided data will bias the model towards this kind of data. This is a reason for the need to include as many corner cases as possible in the data. Biasing can easily happen when you have too few data to provide a statistically well founded training.

The gorilla example may sound fun but it can also upset people and in some cases the wrong descision can even cause inrepairable damage. Google's quick fix to the problem was to remove the gorilla category from their classifyer. This approach may work for a trivial service like picture categorization tool, but yet again what if it is an essential category for the model?

The solution was to remove Gorilla from the category

### 5.1.1 A better solution to avoid biasing

**Use training sets with more diverse people**

The better solution to avoid biasing mistakes is to use a large data base with more variations one example is the IBM Diverse Face Dataset. This face dataset not only provides great variation in people but also adds features to categorize the pictures even further. The figure below shows some samples from the face dataset with categories like:

- Accessories like eyeglases and hats
- Different hair styles
- Face shapes
- Face expressions

IBM Diverse Face Dataset


Fig. 5.1: Mistakes that can happen due bias caused by insufficent training data.

## 5.2 Image data and labels

In the prevoius example with face pictures we started to look into categories of pictures. These pictures were provided with labels bescribing the picture content. The next dataset we will look at is the MNIST data set, which we already have seen a couple of times in this lecture.

In the example below we have extracted the numbers 1,2, and 3. The histogram to the right shows the distribution of the numbers in the extracted data set.

```
(img, label), _ = mnist.load_data()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 4))
d_subset = np.where(np.in1d(label, [1, 2, 3]))[0]

# Visualization
ax1.imshow(montage2d(img[d_subset[:64]]), cmap='gray'), ax1.set_title('Images'), ax1.
↪axis('off')
ax2.hist(label[d_subset[:64]], np.arange(11)),          ax2.set_title('Digit␣
↪Distribution');
```

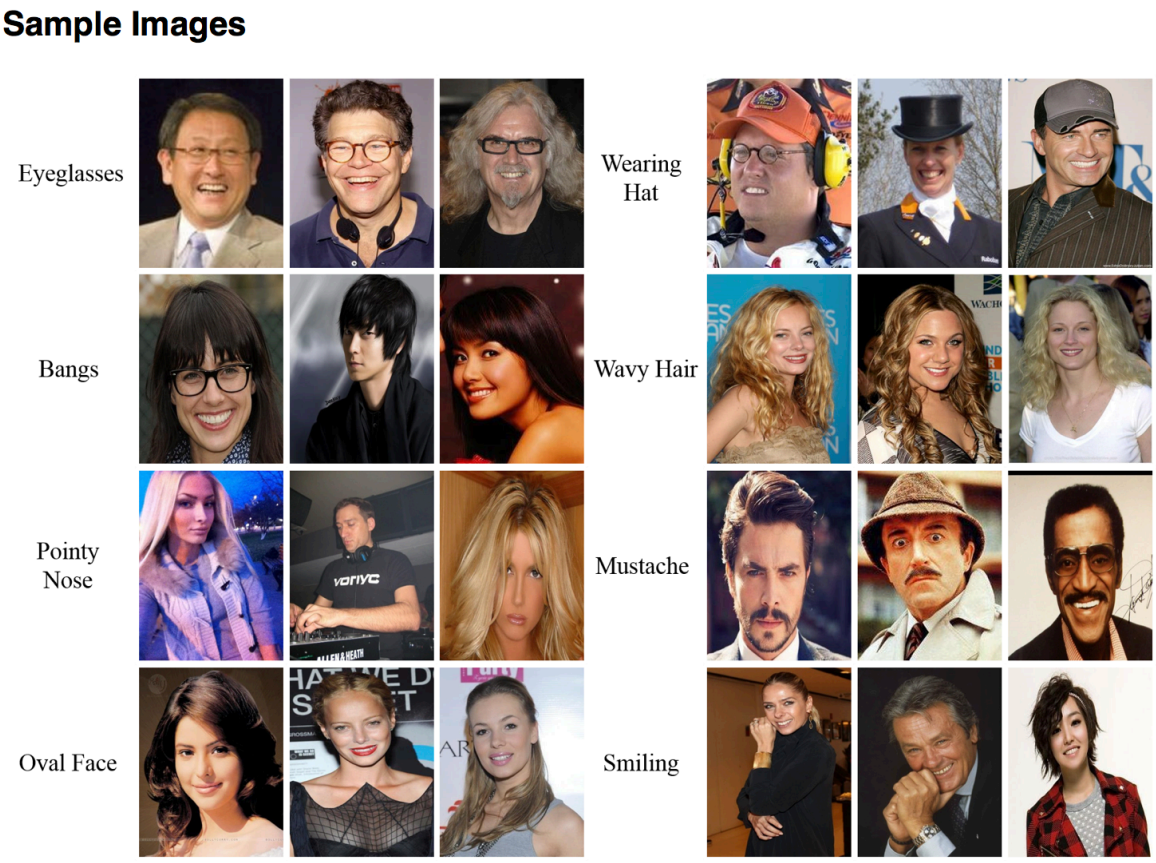
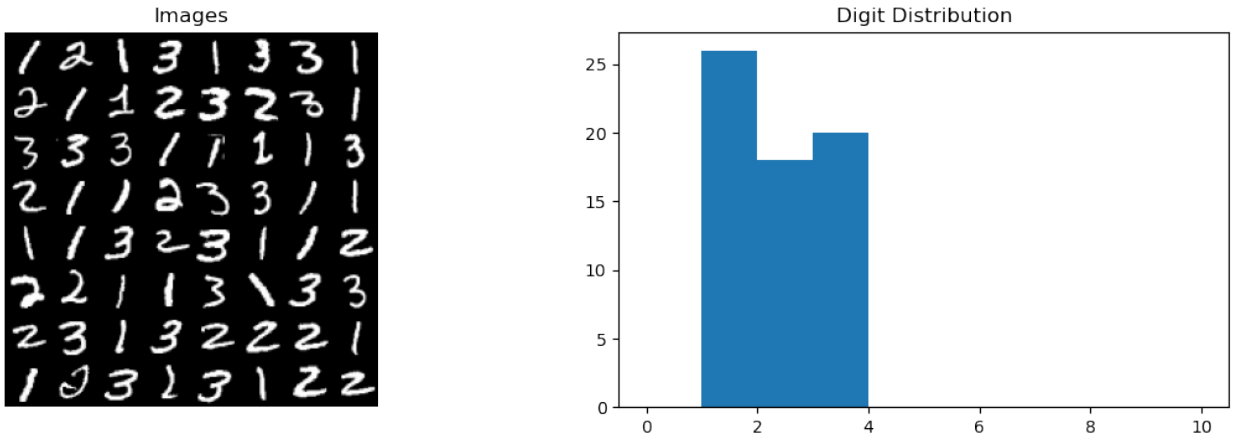



Fig. 5.2: Use a database with more diverse people to avoid biasing.




## 5.3 Limited data

Machine learning methods require a lot of training data to be able to build good models that are able to detect the features they are intended to.

*Different types of limited data*:

- Few data points or limited amounts of images

This is very often the case in neutron imaging. The number of images collected during an experiment session is often very low due to the long experiment duration and limited amount of beam time. This makes it hard to develop segmentation and analysis methods for single experiments. The few data points problem can partly be overcome by using data from previous experiment with similar characteristics. The ability to recycle data depends on what you want to detect in the images.

- Unbalanced data

Unbalanced data means that the ratio between the data points with features you want detect and the total number data points is several orders of magnitude. E.g roots in a volume like the example we will look at later in this lecture. There is even a risk that the distribution of the wanted features is overlapped by the dominating background distribution.


Fig. 5.3: Two cases of unblanaced data; (a) the classes are well separated and the feature class is clearly visible in the tail distribution of the background and (b) the feature class is embeded in the background making it hard to detect.

Case (a) can most likely be segmented using one of the many histogram based thresholding methods proposed in literature. Case (b) is much harder to segment as the target features have similar gray levels as the background. This case requires additional information to make segmentation posible.

- Little or missing training data

A complete set of training data contains both input data and labelled data. The input data is easy to obtain, it is the images you measured during your experiment. The labelled data is harder to get as it is a kind of chicken and egg problem. In

particular, if your experiment data is limited. In that case, you would have to mark-up most of the available data to obtain the labeled data. Which doesn't make sense because

- then you'd already solved the task before even starting to train your segmentation algorithm.
- An algorithm based on learning doesn't improve the results, it only make it easier to handle large amounts of data.

CHAPTER

SIX

# AUGMENTATION

Obtaining more experiment data is mostly relatively hard,

- Time in the lab is limited.
- Sample preparation is expensive.
- The number of specimens is limited.

Still, many supervised analysis methods require large data sets to perform reliably. A method to improve this situation is to use data augmentation. This means that you take the existing data and distorts it using different transformations or add features.

- Most groups have too little well-labeled data and labeling new examples can be very expensive.
- Additionally there might not be very many cases of specific classes.
- In medicine this is particularly problematic, because some diseases might only happen a few times in a given hospital and you still want to be able to recognize the disease and not that particular person.

## 6.1 Typical augmentation operations

### 6.1.1 Transformations

- Shift
- Zoom
- Rotation
- Intensity
- Normalization
- Scaling
- Color
- Shear

### 6.1.2 Further modifications

- Add noise
- Blurring

## 6.2 Some augmentation examples

The figure below shows some examples of augmentations of the same image. You can also add noise and modulate the image intensity to increase the variations further.

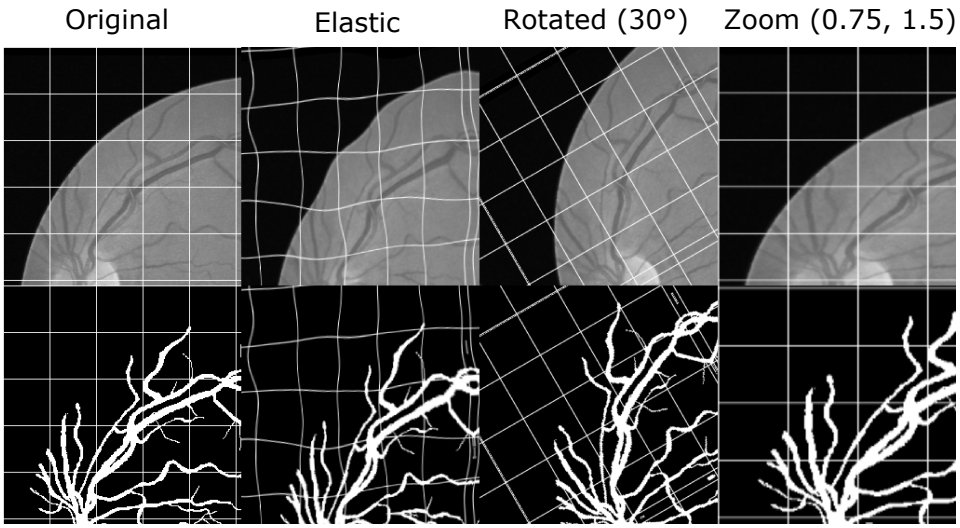



Fig. 6.1: A retinal image modified using different augmentation techniques (source: https://drive.grand-challenge.org/DRIVE/) prepared by Gian Guido Parenza.

Retial images from DRIVE prepared by Gian Guido Parenza.

## 6.3 Limitations of augmentation

- What transformations are normal in the images?
- CT images usually do not get flipped (the head is always on the top)
- The values in CT images have a physical meaning (Hounsfield unit), → scaling them changes the image
- How much distortion is too much?
- Can you still recognize the features?

## 6.4 Keras ImageDataGenerator

Help page of the data generator

```
ImageDataGenerator(
    ['featurewise_center=False', 'samplewise_center=False', 'featurewise_std_
 ↪normalization=False', 'samplewise_std_normalization=False', 'zca_whitening=False',
 ↪'zca_epsilon=1e-06', 'rotation_range=0.0', 'width_shift_range=0.0', 'height_shift_
 ↪range=0.0', 'shear_range=0.0', 'zoom_range=0.0', 'channel_shift_range=0.0', "fill_
 ↪mode='nearest'", 'cval=0.0', 'horizontal_flip=False', 'vertical_flip=False',
 ↪'rescale=None', 'preprocessing_function=None', 'data_format=None'],
)
Docstring:
Generate minibatches of image data with real-time data augmentation.

# Arguments
    featurewise_center: set input mean to 0 over the dataset.
    samplewise_center: set each sample mean to 0.
    featurewise_std_normalization: divide inputs by std of the dataset.
    samplewise_std_normalization: divide each input by its std.
    zca_whitening: apply ZCA whitening.
    zca_epsilon: epsilon for ZCA whitening. Default is 1e-6.
    rotation_range: degrees (0 to 180).
    width_shift_range: fraction of total width, if < 1, or pixels if >= 1.
    height_shift_range: fraction of total height, if < 1, or pixels if >= 1.
    shear_range: shear intensity (shear angle in degrees).
    zoom_range: amount of zoom. if scalar z, zoom will be randomly picked
        in the range [1-z, 1+z]. A sequence of two can be passed instead
        to select this range.
    channel_shift_range: shift range for each channel.
    fill_mode: points outside the boundaries are filled according to the
        given mode ('constant', 'nearest', 'reflect' or 'wrap'). Default
        is 'nearest'.
        Points outside the boundaries of the input are filled according to the given␣
 ↪mode:
            'constant': kkkkkkkk|abcd|kkkkkkkk (cval=k)
            'nearest':  aaaaaaaa|abcd|dddddddd
            'reflect':  abcddcba|abcd|dcbaabcd
            'wrap':  abcdabcd|abcd|abcdabcd
    cval: value used for points outside the boundaries when fill_mode is
        'constant'. Default is 0.
    horizontal_flip: whether to randomly flip images horizontally.
    vertical_flip: whether to randomly flip images vertically.
    rescale: rescaling factor. If None or 0, no rescaling is applied,
        otherwise we multiply the data by the value provided. This is
        applied after the `preprocessing_function` (if any provided)
        but before any other transformation.
    preprocessing_function: function that will be implied on each input.
        The function will run before any other modification on it.
        The function should take one argument:
        one image (Numpy tensor with rank 3),
```

### 6.4.1 A Keras ImageDataGenerator example

There are quite many degrees of freedom to use the ImageDataGenerator. The generator is given all boundary condition at initialization time. Below you see an example of how it can be initialized.

```
from keras.datasets import mnist
```

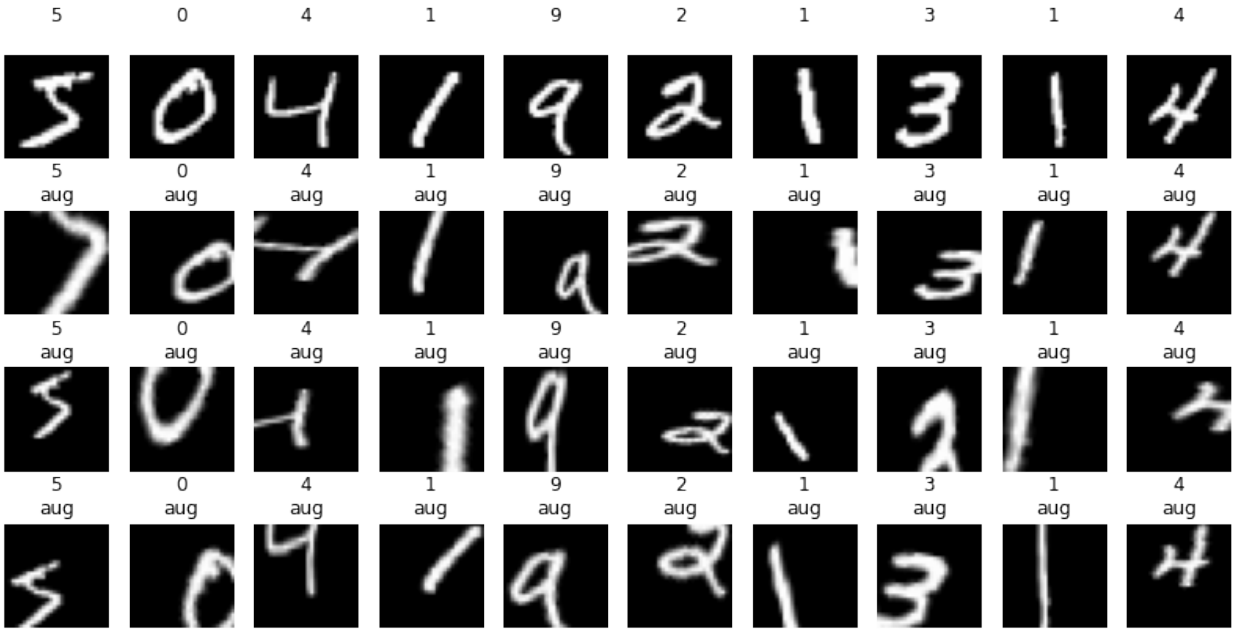
```
from keras.preprocessing.image import ImageDataGenerator

img_aug = ImageDataGenerator(
    featurewise_center  = False,
    samplewise_center   = False,
    zca_whitening       = False,
    zca_epsilon         = 1e-06,
    rotation_range      = 30.0,
    width_shift_range   = 0.25,
    height_shift_range  = 0.25,
    shear_range         = 0.25,
    zoom_range          = 0.5,
    fill_mode           = 'nearest',
    horizontal_flip     = False,
    vertical_flip       = False
)
```

## 6.5 Augmenting MNIST images

Even something as simple as labeling digits can be very time consuming (maybe 1-2 per second).

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
%matplotlib inline
```

```
(img, label), _ = mnist.load_data(); img = np.expand_dims(img, -1)
fig, m_axs = plt.subplots(4, 10, figsize=(14, 7))
# setup augmentation
img_aug.fit(img)
real_aug = img_aug.flow(img[:10], label[:10], shuffle=False)
for c_axs, do_augmentation in zip(m_axs, [False, True, True, True]):
    if do_augmentation:
        img_batch, label_batch = next(real_aug)
    else:
        img_batch, label_batch = img, label
    for c_ax, c_img, c_label in zip(c_axs, img_batch, label_batch):
        c_ax.imshow(c_img[:, :, 0], cmap='gray', vmin=0, vmax=255)
        c_ax.set_title('{}\n{}'.format( c_label, 'aug' if do_augmentation else '')),␣
↪c_ax.axis('off');
```




## 6.6 A larger open data set

We can use a more exciting dataset to try some of the other features in augmentation.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Here are some examples from the CIFAR10 dataset

### 6.6.1 Augmenting CIFAR10 images

```
from keras.datasets import cifar10
(img, label), _ = cifar10.load_data()
```

```
img_aug = ImageDataGenerator(
    featurewise_center  = True,
    samplewise_center   = False,
    zca_whitening       = False,
    zca_epsilon         = 1e-06,
    rotation_range      = 30.0,
    width_shift_range   = 0.25,
    height_shift_range  = 0.25,
    channel_shift_range = 0.25,
    shear_range         = 0.25,
    zoom_range          = 1,
    fill_mode           = 'reflect',
    horizontal_flip     = True,
    vertical_flip       = True
)
```

### 6.6.2 Running the CIFAR augmentation

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
%matplotlib inline
```

```
fig, m_axs = plt.subplots(4, 10, figsize=(18, 8))
# setup augmentation
img_aug.fit(img)
real_aug = img_aug.flow(img[:10], label[:10], shuffle=False)
for c_axs, do_augmentation in zip(m_axs, [False, True, True, True]):
    if do_augmentation:
        img_batch, label_batch = next(real_aug)
        img_batch -= img_batch.min()
        img_batch = np.clip(img_batch/img_batch.max() *
                            255, 0, 255).astype('uint8')
    else:
        img_batch, label_batch = img, label
    for c_ax, c_img, c_label in zip(c_axs, img_batch, label_batch):
        c_ax.imshow(c_img)
        c_ax.set_title('{}\n{}'.format(
            c_label[0], 'aug' if do_augmentation else ''))
        c_ax.axis('off')
```

CHAPTER

SEVEN

# BASELINES

- A baseline is
  - a simple,
  - easily implemented and understood model
  - that illustrates the problem
  - and the 'worst-case scenario' for a model that learns nothing (some models will do worse, but these are especially useless).
- Why is this important?

## 7.1 Baseline model example

I have a a model that is >99% accurate for predicting breast cancer:

$$\text{DoIHaveBreastCancer}(\text{Age}, \text{Weight}, \text{Race}) = \text{No!}$$

## 7.2 The dummy classifier

Let's train the dummy classifier with some values related to healthy and cancer sick patients. Measurements values 0,1, and 2 are healthy while the value 3 has cancer. We train the classifyer with the strategy that the most frequent class will predict the outcome.

```
from sklearn.dummy import DummyClassifier

dc = DummyClassifier(strategy='most_frequent')

dc.fit([0, 1, 2, 3],
       ['Healthy', 'Healthy', 'Healthy', 'Cancer'])
```

```
DummyClassifier(strategy='most_frequent')
```

**Testing the outcome of the classifyer**

```
for idx in [0,1,3,100] :
    print('Prediction for {0} is {1}'.format(idx,dc.predict([idx])[0]))
```

```
Prediction for 0 is Healthy
Prediction for 1 is Healthy
Prediction for 3 is Healthy
Prediction for 100 is Healthy
```

With these few lines we test what happens when we provide some numbers to the classifyer. The numbers are

- 0 and 1, which are expected to be healthy
- 3 , which has cancer
- 100, unknown to the model

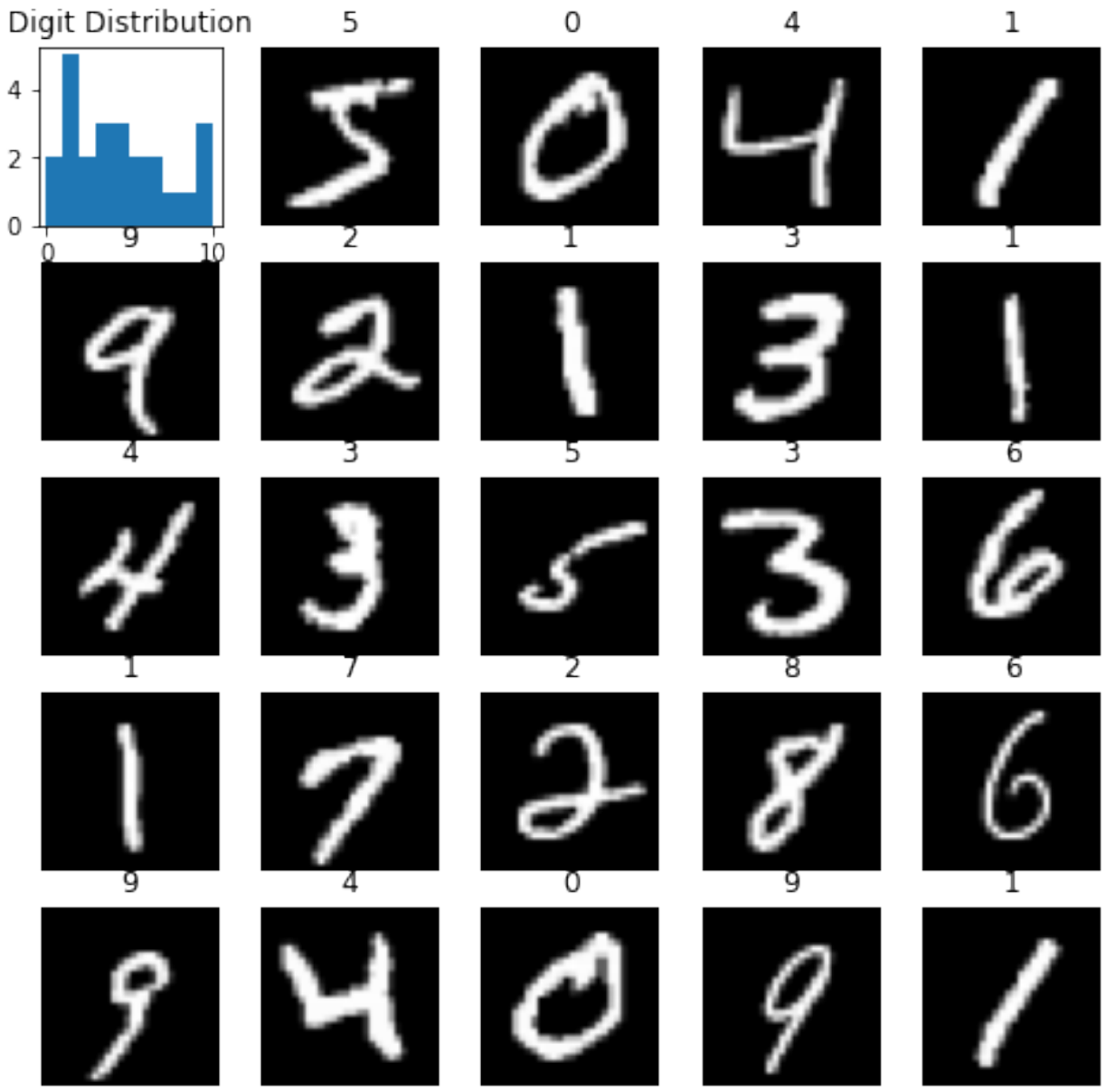
So, the classifyer tells us that all values are from healthy patients… not really good! The reason is that it was told to tell us the category of the majority, which is that the patient is healthy.

## 7.3 Try dummy classifier on MNIST data

The previous basic problem showed us how the dummy classifier work. Now we want to use it with the handwritten numbers in the MNIST dataset. The first step is to load the data and check how the distribution of numbers in the data set using a histogram.

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from skimage.util import montage as montage2d
%matplotlib inline
```

```
(img, label), _ = mnist.load_data()
fig, m_axs = plt.subplots(5, 5, figsize=(8, 8)); m_axs= m_axs.ravel()
m_axs[0].hist(label[:24], np.arange(11)), m_axs[0].set_title('Digit Distribution')

for i, c_ax in enumerate(m_axs[1:]):
    c_ax.imshow(img[i], cmap='gray')
    c_ax.set_title(label[i]); c_ax.axis('off')
```




### 7.3.1 Let's train the model...

Now we want to train the model with our data. Once again we use the *most frequent* model. The training is done in the first 24 images in the data set. The fitting requires that we provide the images with numbers and their associated labels telling the model how to interpret the image.

```
dc = DummyClassifier(strategy='most_frequent')
dc.fit(img[:24], label[:24])
```

```
DummyClassifier(strategy='most_frequent')
```

**A basic test**

In the basic test, we provide the first ten images and hope to get predictions which numbers they represent.

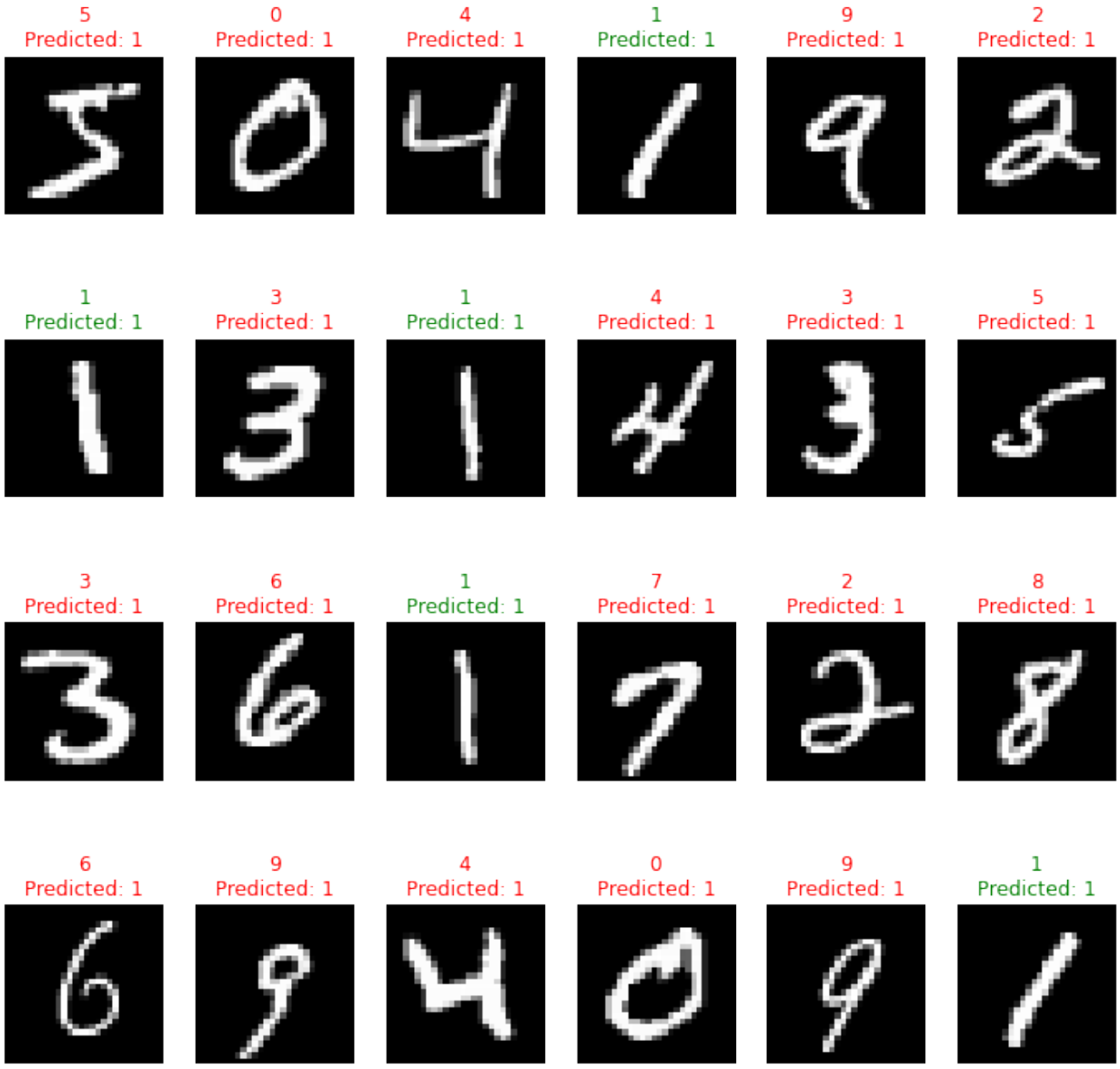
```
dc.predict(img[0:10])
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=uint8)
```

### 7.3.2 Test on the images

Let's see how good these predictions really are by showing the images along with their labels and the prediction of the trained model.

```
fig, m_axs = plt.subplots(4, 6, figsize=(12, 12))
for i, c_ax in enumerate(m_axs.flatten()):
    c_ax.imshow(img[i], cmap='gray')

    prediction = dc.predict(img[i])[0]

    c_ax.set_title('{}\nPredicted: {}'.format(label[i],prediction), color='green' if␣
↪prediction == label[i] else 'red'), c_ax.axis('off');
```

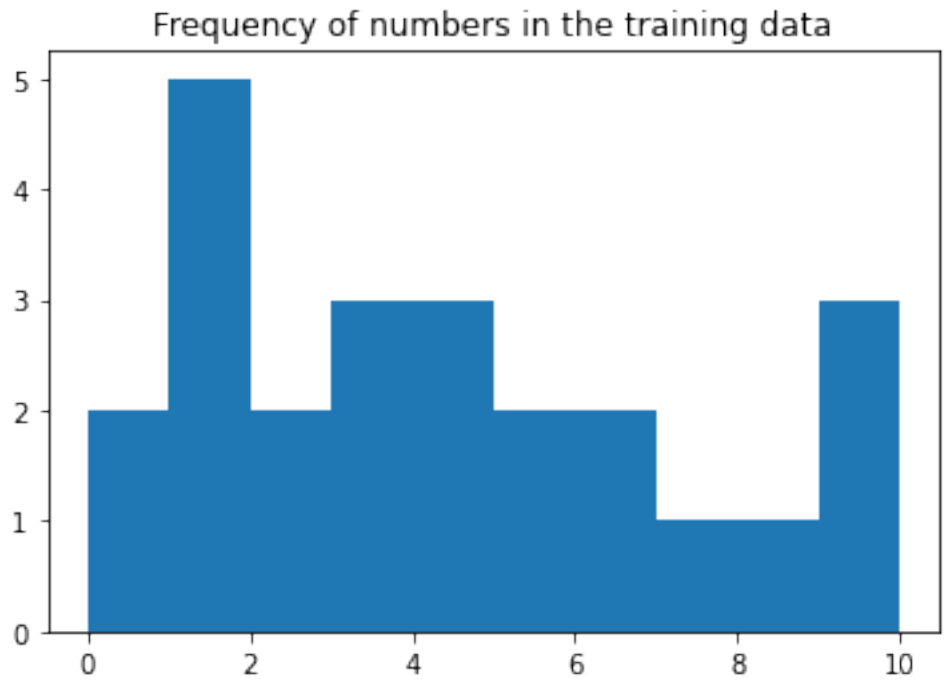



### 7.3.3 ... why are all predictions = 1?

The result of the basic classifyer was quite disapointing. It told us that all ten images contained the number '1'. Now, why is that?

This can be explained by looking at the label histogram:

```
plt.hist(label[:24], np.arange(11)); plt.title('Frequency of numbers in the training␣
↪data');
```




Here, we see that there are most '1's in the training data. We have been using the *most frequent* model for training the classifyer therefore the response '1' is the only answer the model can give us.

## 7.4 Nearest Neighbor

A better baseline

This isn't a machine learning class and so we won't dive deeply into other methods, but nearest neighbor is often a very good baseline (that is also very easy to understand). You basically take the element from the original set that is closest to the image you show.

Figure from J. Russ, Image Processing Handbook

You can make the method more robust by using more than one nearest neighbor (hence K nearest neighbors), but that we will cover later in the supervised methods lecture.

### 7.4.1 Let's load the data again...

Let's come back to the MNIST numbers again. This time, we will try the k-nearest neighbors as baseline and see if we can get a better result than with the dummy classifyer with majority voting.

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from skimage.util import montage as montage2d
%matplotlib inline
```

```
(img, label), _ = mnist.load_data()
fig, m_axs = plt.subplots(5, 5, figsize=(12, 12))
m_axs[0, 0].hist(label[:24], np.arange(11))
```

(continues on next page)

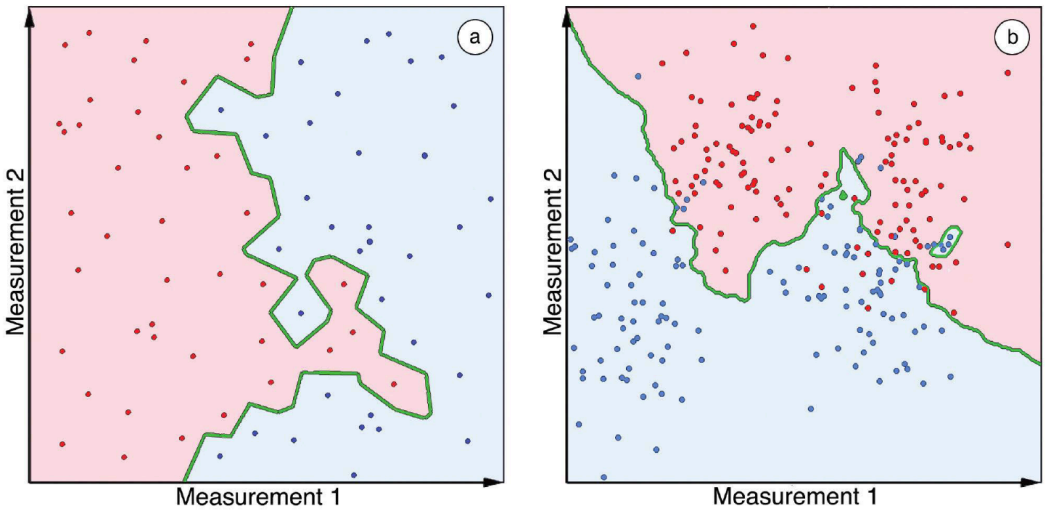
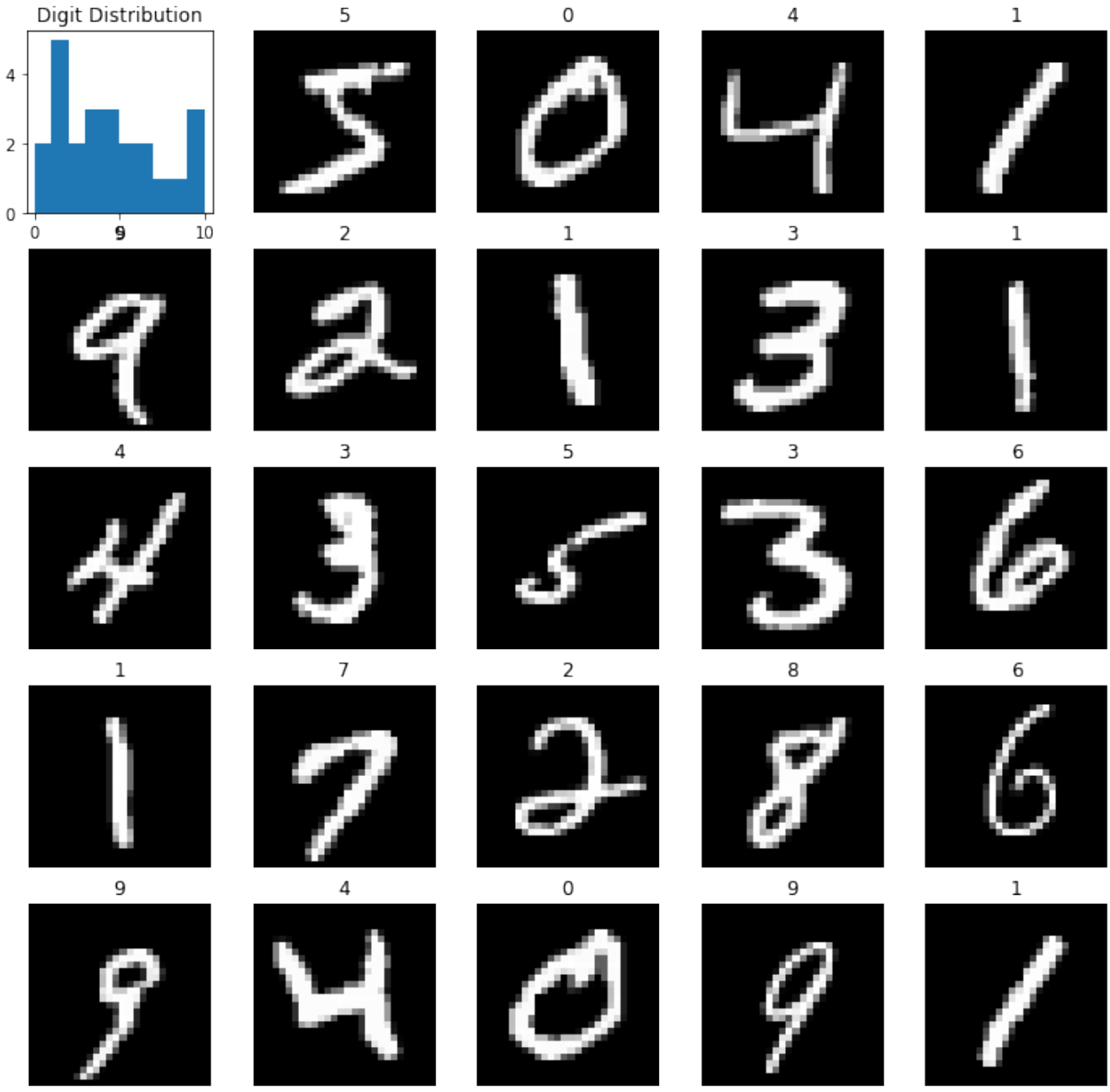



***Figure 12.58*** *The irregular boundary between two classes using k-nearest-neighbor classification: (a) k = 1; (b) k = 9.*

Fig. 7.1: Examples of the k-nearest neighbors classifyer (Figure from J. Russ, Image Processing Handbook).

(continued from previous page)

```
m_axs[0, 0].set_title('Digit Distribution')
for i, c_ax in enumerate(m_axs.flatten()[1:]):
    c_ax.imshow(img[i], cmap='gray')
    c_ax.set_title(label[i]); c_ax.axis('off');
```




### 7.4.2 Training k-nearest neighbors

The training of the k-nearest neigbors consists of filling feature vectors into the model and assign each vector to a class.

But images are not vectors… so what we do is to rearrange the $N \times M$ images into a vector with the dimensions $M \cdot N \times 1$.

```
from sklearn.neighbors import KNeighborsClassifier
neigh_class = KNeighborsClassifier(n_neighbors=1)

N = 24
neigh_class.fit(img[:N].reshape((N, -1)), label[:N])
```

```
KNeighborsClassifier(n_neighbors=1)
```

### 7.4.3 Predict on a few images

The prediction of which class an image belongs to is done by reshaping the input image into a vector in the same manner as for the training data. Now we will compare the input vector $u$ to all the vectors in the trained model $v_i$ by computing the Euclidean distance between the vectors. This can easily be done by the inner product of the two vectors:

$$D_i = (v_i - u)^T \cdot (v_i - u) = scalar$$

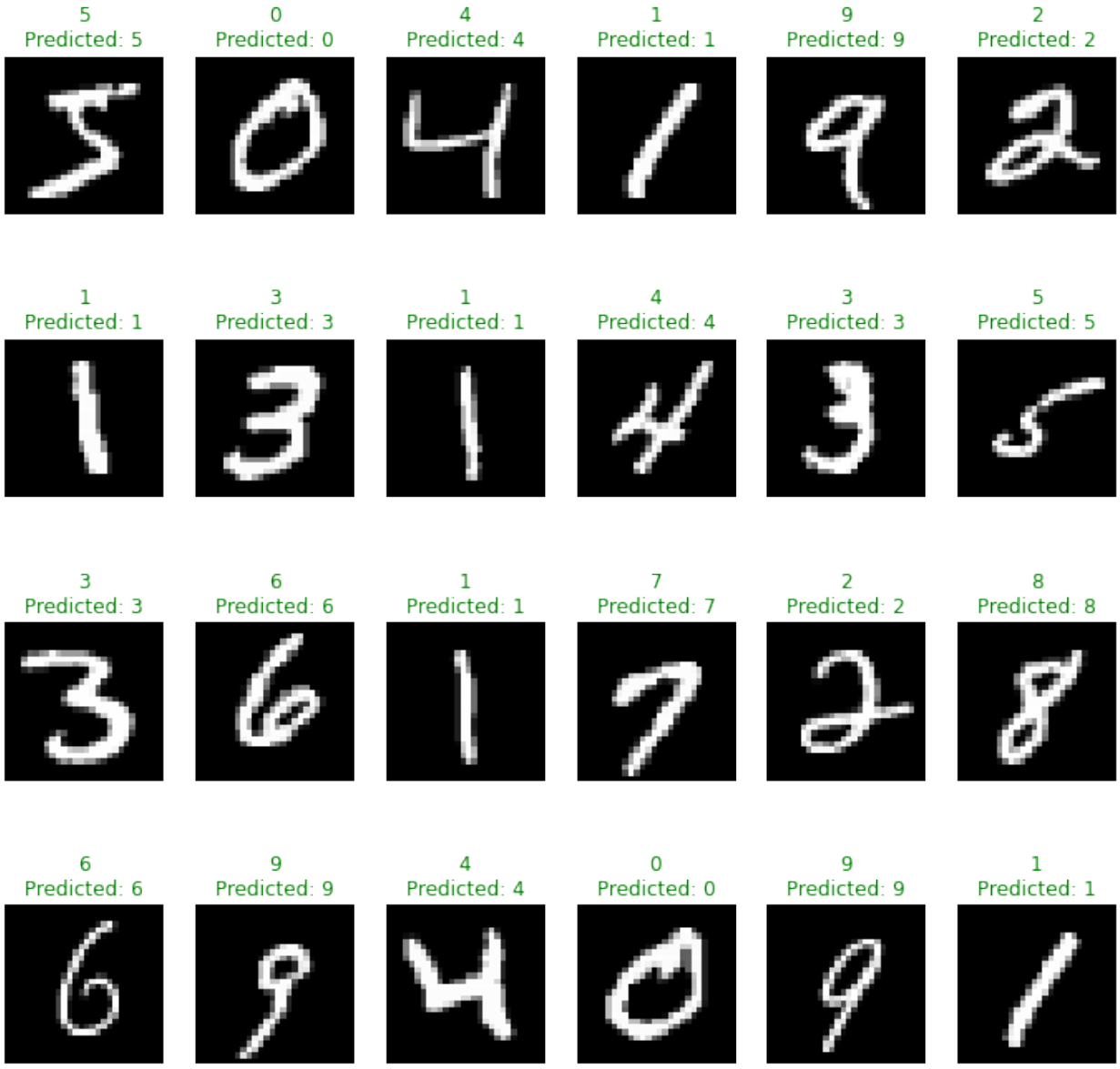
The class is chosen by the model vector that is closest to the input vector, i.e. having the smallest $D_i$. This calculations are done for you as a black box in the `KNeighborsClassifier`, you only have to reshape the images into the right format.

```
neigh_class.predict(img[0:10].reshape((10, -1)))
```

```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4], dtype=uint8)
```

### 7.4.4 Compare predictions with the images

```
fig, m_axs = plt.subplots(4, 6, figsize=(12, 12))
for i, c_ax in enumerate(m_axs.flatten()):
    c_ax.imshow(img[i], cmap='gray')

    prediction = neigh_class.predict(img[i].reshape((1, -1)))[0]

    c_ax.set_title('{}\nPredicted: {}'.format(label[i],prediction), color='green' if␣
↪prediction == label[i] else 'red')
    c_ax.axis('off');
```




Wow, 100% correct!

### 7.4.5 100% for a baseline !!?

Wow the model works really really well, it got every example perfectly.

What we did here (a common mistake) was evaluate on the same data we 'trained' on which means the model just correctly recalled each example. This is natural as there is always an image that gives the distance $D_i = 0$.

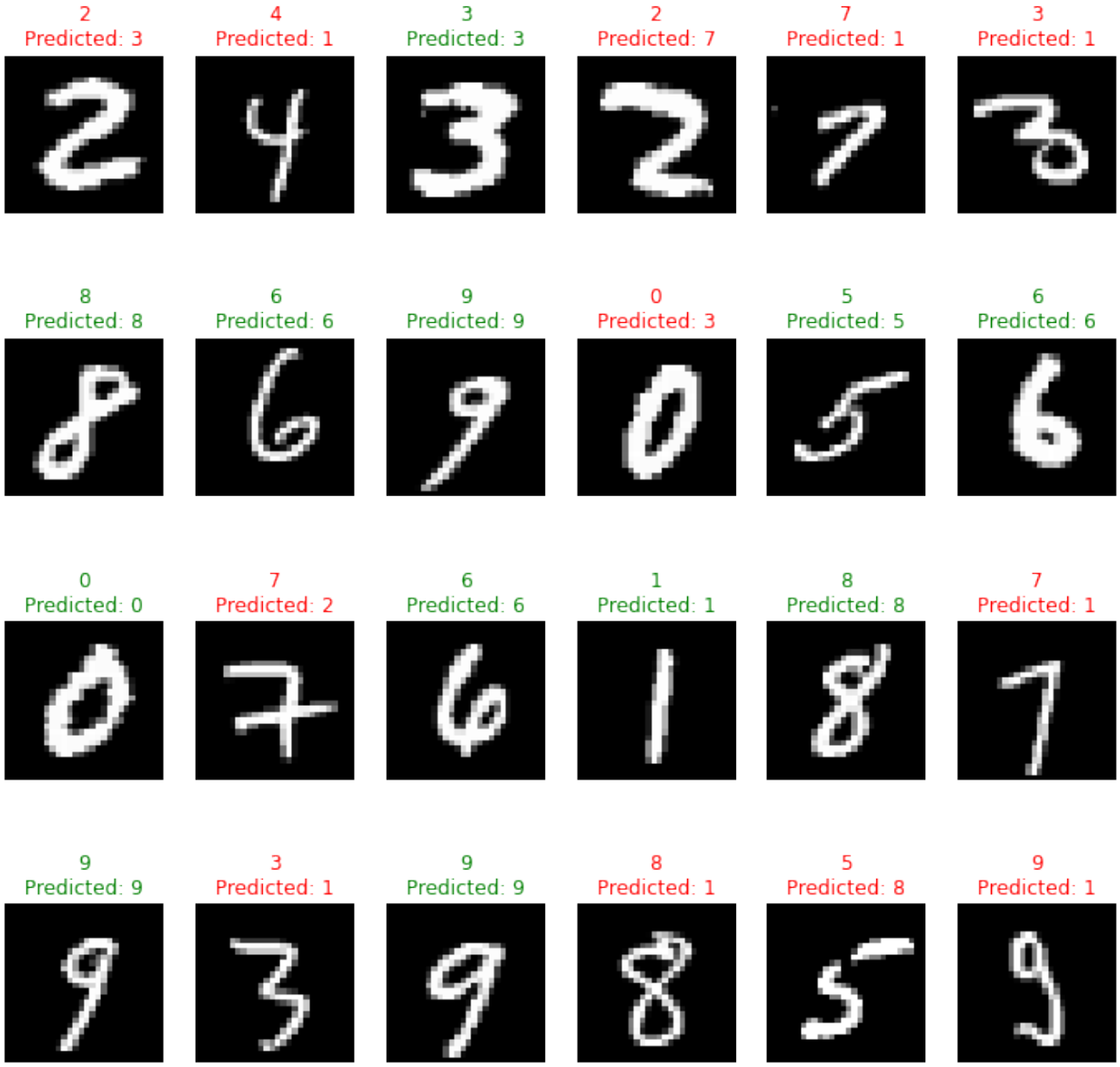
Now, if we try it on new images we can see the performance drop but still a somewhat reasonable result.

```
fig, m_axs = plt.subplots(4, 6, figsize=(12, 12))
for i, c_ax in enumerate(m_axs.flatten(), 25):
    c_ax.imshow(img[i], cmap='gray')
    prediction = neigh_class.predict(img[i].reshape((1, -1)))[0];
```

(continues on next page)

(continued from previous page)

```
    c_ax.set_title('{}\nPredicted: {}'.format(label[i],prediction), color='green' if␣
↪prediction == label[i] else 'red')
    c_ax.axis('off')
```



## 7.5 How good is good?

From the previous example, we saw that the classify doesn't really reach the 100% accuracy on unseen data, but rather makes a mistake here or there. Therefore we need to quantify how good it really is to be able to compare the results with other algorithms. We will cover more tools later in the class but now we will show the accuracy and the confusion matrix for our simple baseline model to evaluate how well it worked.

### 7.5.1 Confusion Matrix

The confusion matrix is a kind of histogram where you count the number of predicted occurances for each actual label. This gives us an idea about the classifyier performance.

We show which cases were most frequently confused

| n=165 | Predicted TRUE | Predicted FALSE |
|---|---|---|
| **Actual TRUE** | *50* | 10 |
| **Actual FALSE** | 5 | *100* |

This is only a simple matrix for the two cases *true* and *false*. The matrix does however grow with the number of classes in the data set. It is always a square matrix as we have the same number of actual classes as we have predicted classes.

### 7.5.2 Confusion matrix for the MNIST classification

We saw that the k-nearest neighbors did a couple of missclassifications on the unseen test data. Now is the question how many mistakes it really does and how many correct labels it assigned. If compute the confusion matrix for this example, we will get a 10x10 matrix i.e. one for each class in the data set.

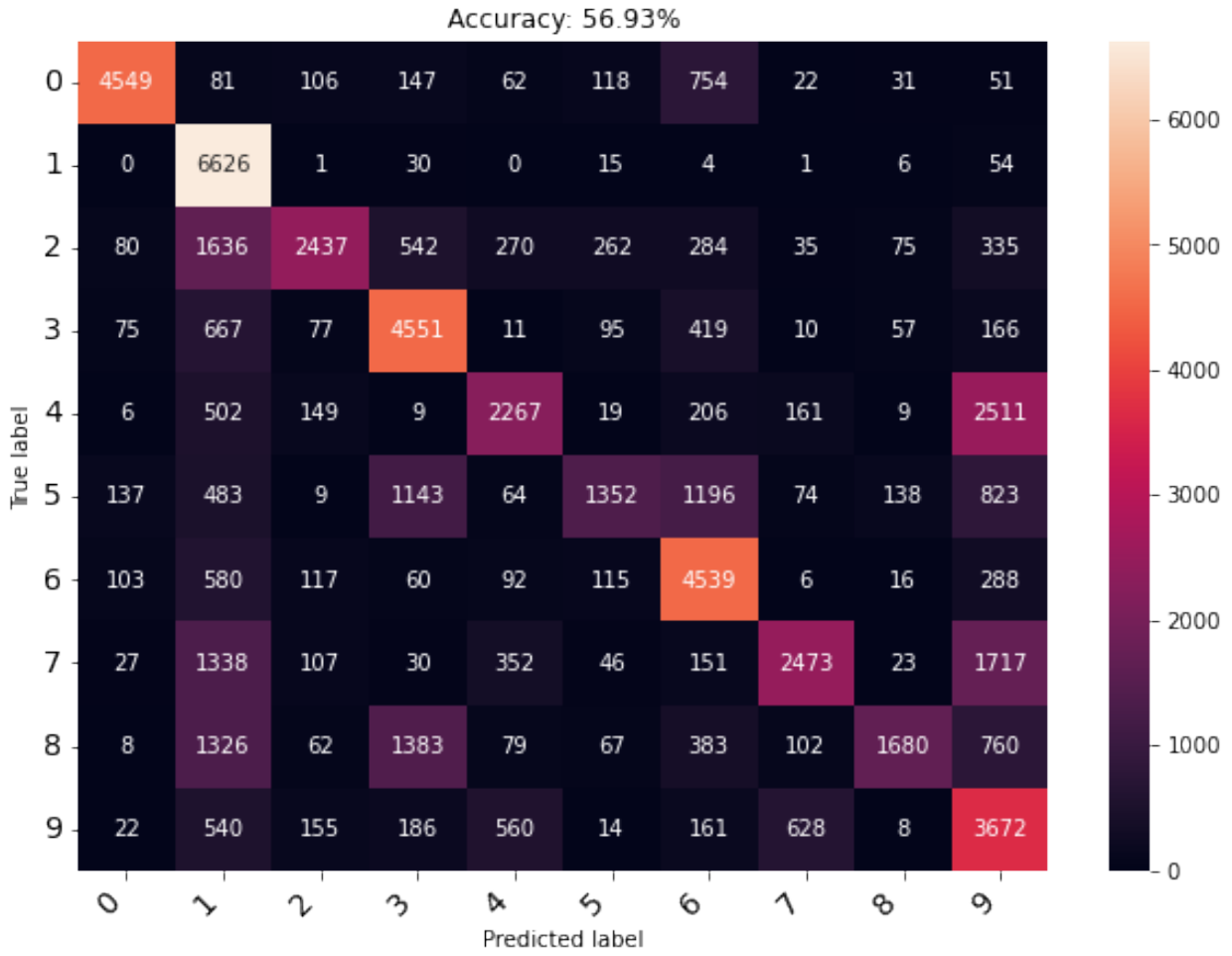
```
import seaborn as sns
import pandas as pd
def print_confusion_matrix(confusion_matrix, class_names, figsize = (10,7),␣
↪fontsize=14):
    """Prints a confusion matrix, as returned by sklearn.metrics.confusion_matrix, as␣
↪a heatmap.

    Stolen from: https://gist.github.com/shaypal5/94c53d765083101efc0240d776a23823

    Arguments
    ---------
    confusion_matrix: numpy.ndarray
        The numpy.ndarray object returned from a call to sklearn.metrics.confusion_
↪matrix.
        Similarly constructed ndarrays can also be used.
    class_names: list
        An ordered list of class names, in the order they index the given confusion␣
↪matrix.
    figsize: tuple
        A 2-long tuple, the first value determining the horizontal size of the␣
↪ouputted figure,
        the second determining the vertical size. Defaults to (10,7).
    fontsize: int
        Font size for axes labels. Defaults to 14.

    Returns
```

(continues on next page)

(continued from previous page)

```
    -------
    matplotlib.figure.Figure
        The resulting confusion matrix figure
    """
    df_cm = pd.DataFrame(
        confusion_matrix, index=class_names, columns=class_names,
    )
    fig, ax1 = plt.subplots(1, 1, figsize=figsize)
    try:
        heatmap = sns.heatmap(df_cm, annot=True, fmt="d")
    except ValueError:
        raise ValueError("Confusion matrix values must be integers.")
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation=0, ha='right
↪', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation=45, ha=
↪'right', fontsize=fontsize)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    return ax1
```

```
from sklearn.metrics import accuracy_score, confusion_matrix
pred_values = neigh_class.predict(img[24:].reshape((-1, 28*28)))
ax1 = print_confusion_matrix(confusion_matrix(label[24:], pred_values), class_
↪names=range(10))
ax1.set_title('Accuracy: {:2.2%}'.format(accuracy_score(label[24:], pred_values)));
```




In this confusion matrix we see that some numbers are easier to classify than others. Some examples are:

- The '0' seems to be hard to confuse with other numers.
- Many images from all categories are falsely assigned to the '1'
- The number '4' is more probable to be assigned a label '9' than the '4'

This experiment was done with a very limited training data set. You can experiment with more neighbors and more training data to see what improvement that brings. In all, there are 60000 images with digits in the data set.

CHAPTER

# EIGHT

# SUMMARY

- The importance of good data
- What is good data
- Preparing data
- Famous data sets
- Augmentation
  - Transformations for increase the data
- Baseline algorithms
  - What is it?
  - How good is our baseline algorithm?
  - The confusion matrix

## 8.1 Next week

Noise and filters