**6. Interesting insights**

**Having a first look** at the data that we were going to work with is very important, there could have been things we would have to take into consideration when creating the code such as the color usage in the images, which kind of body-parts the infections happened to be located at, similarities and differences in the classes. These were different things that could be seen by quick inspection of the dataset. We decided to keep all the eight different classes that the dataset provided us with even though we saw similarities and problems. We were optimistic that we could create models that could distinguish these similarities in the dataset and took this as a challenge.

Since our dataset consisted of only images there were not many other insights, we could get outside from just having a look at the dataset, there would have been a different scenario if the dataset was purely datapoints. Which in that case, different tools could have been used to have a look at the data, such as finding patterns or strong features. In our case, we just had a look at the dataset and kept some of the finds in mind for later.

**7. Data Preprocessing**

**Data preprocessing** is a necessary step when preparing for machine learning, the data that goes in needs to be in the correct format to train and predict. When dealing with data it is important to first get an overview of how the data looks to begin with, then what size and shape the desired data needs to be. The goal of this project was to get an image input, then to predict what kind of disease detected in the image. This meant that we had to figure out how we would have to process image data in a way that the machine learning models accepted.

The first step was experimenting with loading an image using the OpenCV library, this would then load in the image for further processing. Numpy is a powerful library for data processing, and combined with OpenCV we could read and shape the data. The dataset of choice contains large amounts of images, this meant that we had to process one image at a time, then combine all these images into a larger array containing all the images in each class. This is where we began, and we quickly realized that there had to be a better way. This is when we started creating our own DataProcessor class, that would handle all aspects of the data processing.

The DataProcessor class has a lot of functions such as generating X and Y data, generating necessary folders, augmenting data using OpenCV, train-test and validation splitting and visualizing the data. The reason for this choice is flexibility, we wanted a great way of testing different techniques fast, such as generating data from the original dataset and the augmented dataset and the different ways of augmenting the images. Putting the extra work into the data loading we were able to automatically detect classes within the dataset folder and load in the images in the desired format and splitting this into X and Y without having to manually change how different parts of the code worked. Using parameters, we were able to create a powerful class that allowed for easy changes.

When creating this class, we thought of different ways in which the data could be shaped, the data needed to be flattened such that the image data and the label only required one row. This means that every pixel, consisting of 3 values (R,G,B) using images of 224 x 224 would give us a total of 150 528 features per image. The data processing takes these steps:

**Load class folder -> Load image -> Reshape to 224 x 224 -> Flatten Data -> Add Label**

When this process was complete, the function loops over each class and adds all the data to a final array, which could then be used for training. All these classes are detected automatically using the OS library and browsing the different folders in the dataset.

When training the model, we quickly realized that the number of images, and the vast number of features would require a lot of computing power to process, especially when we started augmenting the images. The model training using augmented images ended up taking roughly 1 hour and 40 minutes. This was not optimal when also considering that predicting one image would take close to a minute to process.

This realization made us wonder if there could be a better way to extract the features of the image, without loosing accuracy. After a lot of experimenting, we found a method that we never expected to work but would later be great. Instead of flattening the image we attempted to sum each column in the image such that we would only have one row, this row was then divided by the height of the image leaving us with the average of all the rows in each column into one single row, this data was then flattened such that the RGB values split up into their own columns. This meant that instead of the data having 150528 features, they would rather have 672 features (224 x 3). For comparison, the graph data generated with this technique (image 2) and the original image (image 1) are shown below.

Et bilde som inneholder tekst, Plottdiagram, diagram

Automatisk generert beskrivelseEt bilde som inneholder føtter, tå, barbeint, person

Automatisk generert beskrivelse

Image 2

Image 1

We also knew that there are existing libraries out there for augmenting data, however we wanted to have full control over what happened to these images. This is why we decided to create our own augmentation function with the help of OpenCV. The augmentation function is designed to take in a few parameters such as minimum and maximum brightness, contrast and zoom. The images were also flipped vertically and horizontally before the images were augmented. During this process, it was important for us to focus on variability while keeping the features intact.

**8. Measuring Performance**

When creating different machine learning models, it is important to keep in mind which kind of way the model’s performance will be measured. In some cases, it might be smart to measure how well the model predicts positives also called recall or sensitivity. In some cases, Specificity is a great measure of how well the model correctly predicts negatives.

However, in the case of this project, we found f1-score to be the best way to measure our performance. The f1-score is a combination of both recall and specificity and will see how many correct predictions the model made from the entire dataset.

F1-Score worked great in our project, since the goal was to find how many images the model managed to correctly predict. The way we gathered data for generating performance measurements was that we created a model, trained it and then we used the models predict function to predict a certain number of images. For the graph data that we generated, we used approximately 2000 images (since each prediction was way faster) and for the entire images we predicted around 200 images (due to the slow prediction times). Then we used scikit-learn’s classification\_report function to compare all the predictions to the actual labels of the same images. The output of this function looks something like the table shown in table-1 below:

Et bilde som inneholder tekst, skjermbilde, nummer, Font

Automatisk generert beskrivelse

table-1

As shown in table-1 there is also a column for support, which is how many predictions of the certain class that has been done, in the example we got an accuracy of 0.66 % which was not quite the performance we wanted to get considering the importance of a high accuracy when dealing with patients and disease treatments.

**9. Algorithms Applied**

**K-Nearest Neighbor**

**Convolutional Neural Network**

**Support Vector Machine**