**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 losing 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 1*

*Image 2*

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**

**Convolutional Neural Networks** is a widely used deep learning technique for accurate and advanced machine learning projects. Large companies use CNN’s for creating accurate and efficient deep learning models, some examples include advanced image classification, sign language translation, lip reading algorithms and way more. CNN’s are in many cases used with image data, either frames in a video or standalone images.

The way a Convolutional Neural Network works is by learning patterns within data. With images, the CNN would be able to learn certain patterns and recognize those patterns with the classes. The structure of the network is highly configurable and requires a lot of experience to understand what really happens behind the scenes. The machine learning engineer will define each layer of the network, some layers search for larger patterns within an image, while other layers might search for smaller patterns.

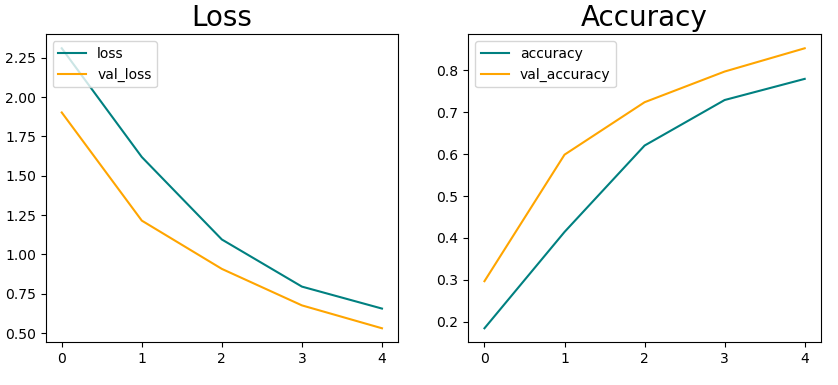
The most common library for such deep learning is TensorFlow, which is developed by Google. An example of how the layers in a convolutional neural network are structured are shown below in (*code example 1*).

There are Conv2D layers, which are smaller portions of the images that will try to learn patterns in that shape, the first Conv2D layer is also called the input layer in our case, where the input shape of the image is specified. The Dropout layers will randomly disable some neurons in the network, this can help with preventing overfitting. The last layer is a dense layer that is specified with 8, which is the number of classes. An activation function, in our case it is the SoftMax activation function which will return a value between zero and one for each class, the class with the highest value is the predicted class.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse *(code example 1)*

When training a CNN, the model will generate training data loss and validation data loss. It then uses these results to adjust the weights in the network to gain a better result in the next epoch. The progress of the training can be plotted with matplotlib to get a visual representation on how the training process went, also to compare the training accuracy versus the validation data accuracy. If the training accuracy ends up being too high, it might be a sign of overfitting, which will result in the model poorly predicting unseen images.



After training our convolutional neural network we achieved a training and validation loss and accuracy as shown in (*graph 1*)

Although the results looked very promising, the model was bias and was almost never able to predict the correct class. Cellulitis was the class that seemed to be bias each time.

*(graph 1)*

To provide an overview of how such a prediction looked, below is a table (Table 2) of all classes and which probabilities each class had to be the prediction image which in this case was an image containing **impetigo:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| cellulitis | impetigo | athlete-foot | nail-fungus | ringworm | cutaneous-larva-migrans | chickenpox | shingles |
| 96,7% | 26,5% | 29,2% | 32,6% | 55,1% | 27,1% | 17,1% | 17,1% |

(Table 2)

We knew that this exam allowed the use of preexisting base models, which could have been done for this exact purpose. ResNet50 is a popular CNN model architecture consisting of 50 layers (<https://datagen.tech/guides/computer-vision/resnet-50/>), which has been proven to work well with this dataset, this can be seen in a notebook linked within the dataset page (<https://www.kaggle.com/code/tantranduc/skin-disease-resnet50-acc-97>). By using the ResNet50 structure, this user was able to get an accuracy of 97%.

The reason we stayed away from this solution was that we wished to create something on our own instead of using a preexisting structure that is known for great results. Experimenting and learning was our main objective of attempting to use convolutional neural networks, and using a preexisting structure would give great results quickly, which was not necessarily our goal.

When stepping into this project, we wanted to learn more about CNN’s even though it is not a part of this course. We learned a lot about how the Convoutional Neural Networks are built and how they operate. However, we had a hard time getting good results with the little knowledge we had at the time. We managed to properly load in the data using the TensorFlow data pipeline, define a model structure and run predictions. The results of the CNN training seemed very promising, however when predicting images, we never got good results. In most cases, the model overfitted and was very bias in one of the classes.

We are sure that with proper knowledge and experience with convolutional neural networks, it would be the best possible model in this projects case. It has been proven to give very good results, even on data with smaller details. The model structure must be engineered and by trial and error, to eventually get the wanted results.

**Support Vector Machines** are a collection of different machine learning techniques, which includes support vector regressor, support vector classifier. These methods can be used for different projects, depending on the goal of the machine learning model. In our case, the support vector classifier is used since we aimed to classify different diseases within 8 different classes. The regressor is used in cases where the model must predict a value instead of a class.

The support vector classifier is a powerful machine learning model, which can accurately learn and predict new data. The algorithm is also one of the faster ones when it comes to prediction times. The standard parameters provided with the **SVM** usually gives good results, however it is possible to tune the model to give even better results, for this we decided to use the GridSearchCV functions from SkLearn.

The best way to understand how the support vector classifier works is to see 2d data classified by the classifier. This also allows for easier understanding on how the different kernels works. Kernels are mathematical equations that decides on how the model will classify, the SVC has four different kernels, which are linear, polynomial, rbf and sigmoid. Which one used depends on what is required from the project. For simplicity there is an image illustration below (*image 3*) which displays three of the different kernels.

Et bilde som inneholder tekst, hest, skjermbilde

Automatisk generert beskrivelse

*Image 3 (*[*https://scikit-learn.org/stable/modules/svm.html*](https://scikit-learn.org/stable/modules/svm.html)*)*

these images display how the model would separate the iris dataset, comparing sepal width and sepal length depending on which kernel is being used. This illustration shows how the kernels would classify a 2d dataset and would create a line separating the different classes. If the data has more than two dimensions, the line would be called a hyperplane, which would be like a piece of paper shaped to separate the different classes in multidimensional space.

The way that the support vector classifier generates these hyperplanes is by calculating where there is the most separation between the datapoints, this is quite efficient since the algorithm only must consider the datapoints that are the closest to this separation as shown in the image below (*image 4*).

Et bilde som inneholder skjermbilde, line, diagram, Plottdiagram

Automatisk generert beskrivelse

This separation in the data will be the decision boundaries for the classification, as shown in this image the two orange datapoints and the blue datapoint form what is called a **support vector**, which allows the algorithm to calculate the shape of the hyperplane.

*image 4* (<https://scikit-learn.org/stable/modules/svm.html>)

During our testing and evaluation we found the support vector classifier to do a great job on the skin disease dataset, we gathered a few different results depending on if the training was done on the original dataset, augmented dataset and graph dataset from both of these which we generated within our DataProcessor class using the generate\_xy\_by\_axis() function. To summarize the results, we gathered are shown in the table below (*table 3*)

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Graph Data / Image Data** | **Accuracy** |
| Original | Image Data | 56% |
| Augmented | Image Data | 77% |
| Original | Graph Data | 34% |
| Augmented | Graph Data | 49% |

*table 3*

In this table there is one model that did particularly well, which is the graph data on the augmented dataset. The result was not achieved using the standard svc hyperparameters, these hyperparameters were tuned using GridSearchCV in the SciKit-learn library. This allows for the machine learning engineer to create a dictionary with hyperparameters. This dictionary is then used to generate different folds of parameters that will be trained and evaluated. The hyperparameters that were tuned in our model was C and gamma, by tuning these we increased these results by approximately 20%. The reason we decided to only use GridSearchCV on the graph data was considering the computing time it would have taken if we used the entire image data consisting of around 150 000 parameters, compared to the graph images with 672 parameters.

We found the support vector classifier to work better than expected on image data considering the similarities in the different classes. The simplicity of using the support vector classifier also allows for quick prototyping and great results.

**10. Discussion**

**Evaluating the models and the workflow with each.**

Throughout this project, we had many different experiences with different data structures and machine learning models. After experimenting and exploring models and parameters we found different results which were both expected and surprising. The *DataProcessor* class that we created was a great choice since we were able to test different data processing techniques to see which kinds of results we could get, especially with the sum of the rows or columns, we were very surprised of the results generated. Not only did we get surprising results with this method of processing images, but we also got incredible training and prediction speed.

When it comes to difficulty, the *convolutional neural network* requires the most knowledge out of the three models that we decided to use. From the data processing to the evaluation, convolutional neural networks need proper knowledge to understand what kind of layers, functions and hyperparameters to use to get the best possible results. Even though the CNN came with it own set of problems and difficulties, we had a great experience experimenting and learning from it, which could be very useful for later development.

The *Support Vector Machines* and *K-Nearest Neighbor*were both quite similar when it comes to difficulty. Both models are similar in the way that they are trained and how predictions work. However, the support vector machines allow for a more structured way of testing hyperparameters with the *GridSearchCV* functions which means that finding the optimal model parameters was simpler and gave greater results in the end. The K-Nearest Neighbor does not have such functions and has fewer hyperparameters to change to get different results. We found the default parameters to work the best for our dataset.

The support vector machine gave the most accurate predictions after searching for the optimal hyperparameters with *GridSearchCV* with an accuracy of 80%. This was quite surprising, considering the model was using the sum of rows data pre-processing. This made us wonder what accuracy the model would get by using the whole image data instead, doing so would have taken a very long time, a rough estimate would be 4-6 days depending on how many different combinations of parameters tested. Which is the exact reason we did not have the ability to do this for this project, however if used in a real-world scenario, this could be the solution to finding a good enough model to be viable for use in the medical field.

The field that the K-Nearest Neighbor excelled in was speed. The training and prediction speeds with KNN was surprising, and even though there were less parameters to change, we still got great results from both the sum of rows data and the entire image data.

**Size of dataset and variation in the data**

Sometimes less is more, but was this the case for this project? There are many ways to gather, process and augment data. The size of the dataset could have an influence on how well the models generalize and learn. We used our own augmentation pipeline, which allowed us to have full control over what kind of transformations happened to the data. We also got it proven to us that the augmented date gave quite a lot better results compared to only training on the original dataset. Using that information, we attempted to increase the number of images generated from the augmentation function, and the results almost stayed the same but the training and prediction time increased. We found the number of images that gave generally the best results and still tried to keep training and prediction time down as much as possible. When it comes to augmentation, there are many ways to augment the images. Which brought the question if we could have transformed the images in a different way to gain even better results.

Considering the variation of the data was something evaluated in the earlier stages of the project. We saw which kind of diseases we were going to work with, one important thing that we had to keep in mind was where these diseases were located on the body. Some of the diseases were specific to certain locations of the body, however some of the other diseases did not have a specific area. we had to keep in mind that the models could learn the different body-parts instead of the disease itself. In the end we decided to use the data that we had available due to difficulty finding more images that we had the right to use. This could have been expanded upon if it would have been taken into use by a hospital. Gaining rights to a larger dataset could allow for manual selection of data with the variations that would give a more generalized and accurate result.

**Saving and loading the models**

Training machine models can take a long time, and often it can be quite smart to save the models to a file that can quickly be loaded for use. *Pickle* is a standard library included in python (<https://docs.python.org/3/library/pickle.html>). Using Pickle, it is possible to save any kind of object to a binary file, this object can then be loaded in the same state as when saved.

By using Pickle, we were able to train a model and save the model object to a file. However, did not expect the file to reach a file-size of almost 10-15 gigabytes. Which made the file very difficult to move from one device to another. There are very few services that allows for transfer of files of this size, from these, almost none of them are free. This was also one of the main reasons we managed to find the sum of rows technique to use less parameters and reduce the size of the final model.

For this reason, we made sure that the dataset has the right size so that it would be possible for each model to take a maximum of two hours to train. Even though we did not end up using pickle too many times to save these models. It would have been done if the model had to be transferred to a customer, which could have been done by using a paid service or by transferring via hardware, such as a memory stick or a hard drive.