Machine Learning Engineer Nanodegree

Capstone Project

Dominik Schniertshauer April 6th, 2020

I. Definition

Project Overview

I'm participating in the *Plant Pathology 2020* challenge hosted on Kaggle. The following descriptions are based on the information shown on Kaggle.

Many plant diseases can be detected by visual inspection. An early detection is necessary to avoid further spread and avoid economic or environmental impact. As with other vision based challenges, deep learning seems to be a promising tool to increase the level of accuracy in detecting and classifying diseases while minimizing human efforts through automation.

Problem Statement

The Kaggle challenge wants participants to develop an algorithm that can automatically dinstinguish between a healthy and non-healthy plant. The class non-healthy is furthermore distinguished into three subclasses, namely rust, scab and multiple diseases (which means both rust and scab). Rust and scab are diseases caused by specific funghi invading trees and in general detectable by the human eye. Therefore it's assumed that the inspection of such diseases can be automated with the help of Machine Learning.

The main challenge I see in this competition is the lack of training data. With only 1800 training images it is required to implement an algorithm that can detect those diseases in 1800 test images.

To approach this problem, I first implemented a baseline model. This baseline model consists of three steps. First, small patches will be extracted from each image. Based on those patches, a dimensionality reduction algorithm is trained to learn basic features from the data. Those learnt features are then extracted from every image and used in a Random Forest classifier, trying to predict the target class.

In a next step, I implemented the final model with the help of Deep Learning. To tackle the lack of training data, I could either look for external training data or use techniques such as transfer learning and data augmentation. I choose the second way and will therefore try multiple algorithms such as **VGG16** or **DenseNet** while increasing the amount of training data by using augmentation techniques.

Metrics

Since I want to participate in the Kaggle competition, I will use their evaluation metric as stated in the competition's description: the mean column-wise AUROC. For each target class the prediction result will be used to calculate the AUROC and then the mean over all columns' AUROC will be used. The column-wise extension of this metric is necessary, since the ROC is calculated for binary classification problems and we deal with a multi-class problem here.

AUROC definition: The AUROC is defined as "Area Under the Receiver Operating Characteristics". To understand this, we first need to understand the ROC (Receiver Operating Characteristics). The ROC curve compares the True Positive Rate with the False Positive Rate of a binary model in a diagram. The AUROC calculates the area under the resulting curve, where a value of 1 would imply perfect seperability of the target classes and a value of 0.5 would mean, that the model has no prediction power whatsoever.

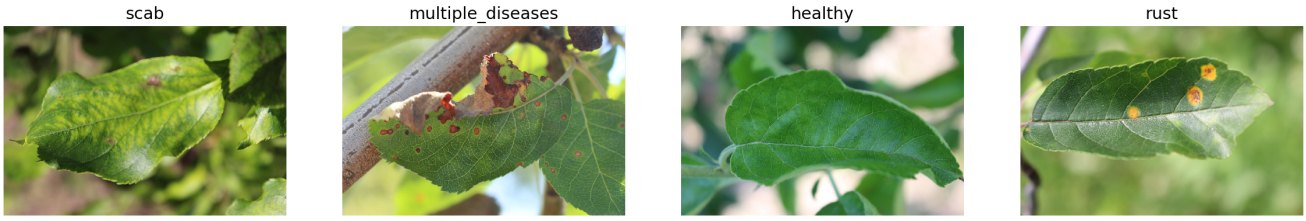
AUROC is to my opinion a great metric for this competition. Our target data is not completely evenly distributed and a measure like accuracy would therefore not reflect the problem space. AUROC in comparison can deal with skewed data and would "punish" competitors that implemented models ignoring the minority class.

II. Analysis

Data Exploration

The dataset consists of exactly 1821 images each for the training and test set. The images show plan leafs that are well centered within the image and of consistent quality. In the training data set we find the three classes **healthy**, **rust** and **scab** more or less evely distributed. However, the target class **multiple_diseases** is underrepresented in the dataset, making up for less than 5% of the dataset.

In the following illustration, you find four examples extracted from the training dataset.



Regarding the disease classes, both rust and scab seem to create visible, oval spots in different colors. By manual inspecting multiple of such pictures, I feel confident to detect the obvious cases by myself. Detecting cases with multiple diseases is more difficult, which only adds up to the challenge of not having a lot of training data for those. Healthy leafs are the most easy to detect and basically characterized by an all green surface.

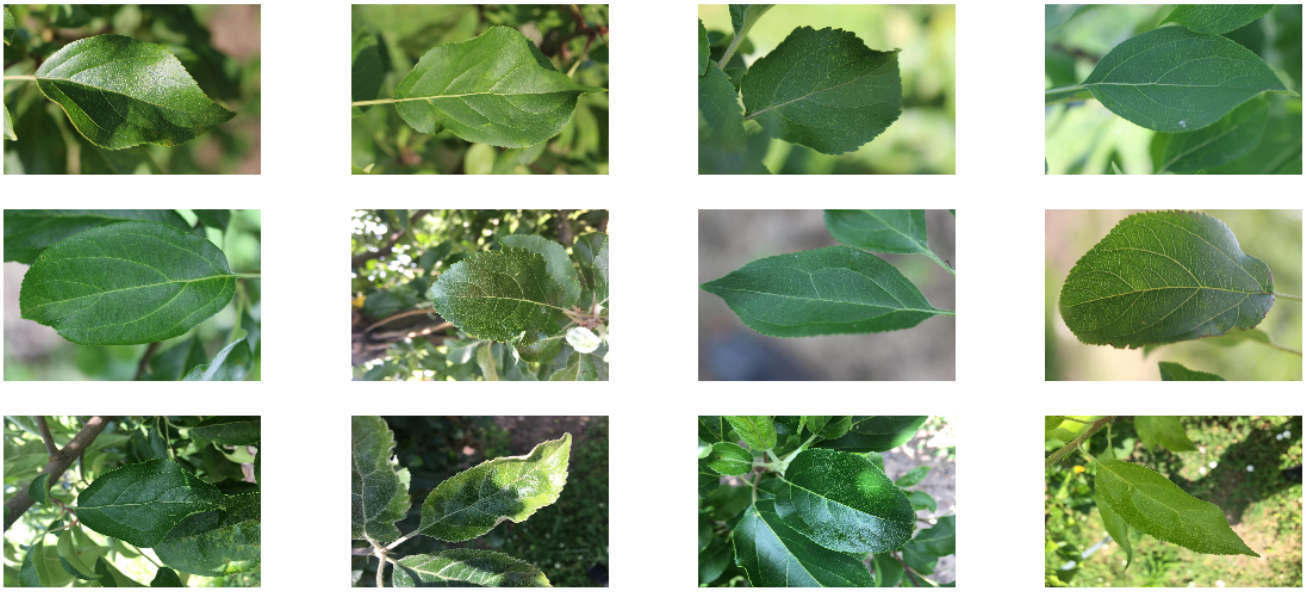
Based on the amount and distribution of the data, I decided to try the following techniques in the next steps: data augmentation, transfer learning and adapting the class weights of the algorithm, to give a higher weight to the minority class while calculating the loss value.

Exploratory Visualization

In this section I want to give you more insights in how the actual training data looks like. First I will show examples out of the actual data to better understand the problem domain. In a next step, I will conduct an analysis of the colorspace with the help of histograms.

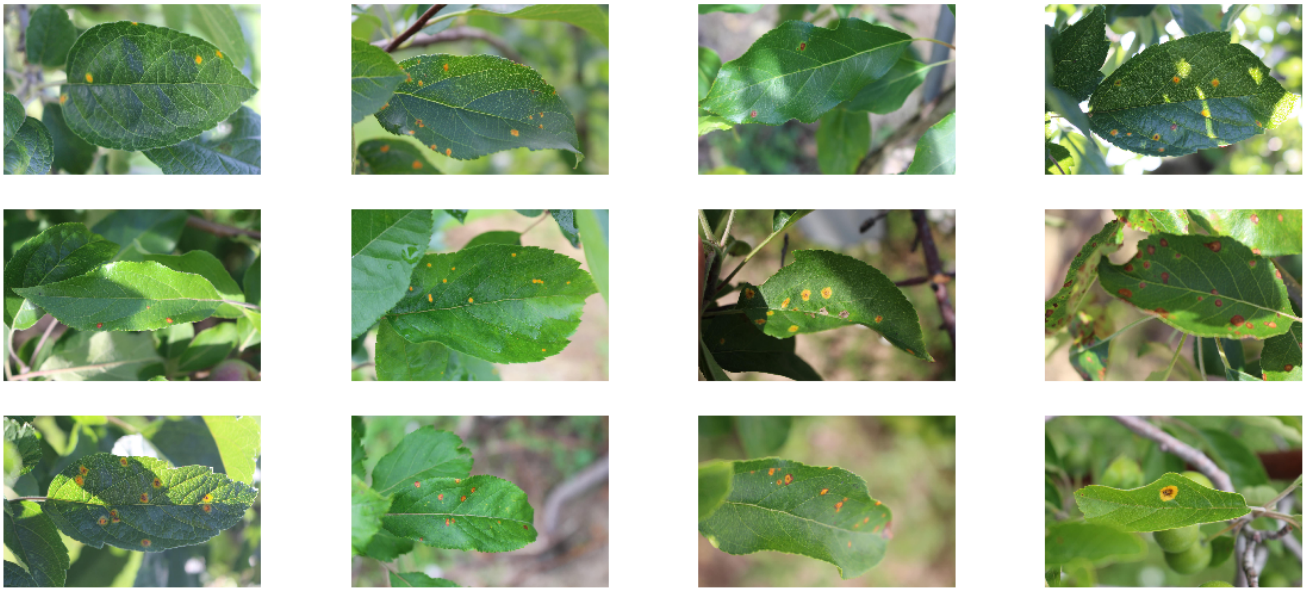
Plants without diseases

Healthy plants are the most easy to detect manually. They are characterized by a flawless green surface, not showing any bigger dark spots.



Plants with rust

Rust leafs seem to be characterized by yellow to brown spots that can be spread all across the leaf surface. The spots ususually have clear edges and also seem to be easy to detect.



Plants with scab

Scab seems a bit harder to be detected by the human eye. There are dark areas on the leafs, but they don't show the clear edges we can see with rust. Therefore on some of the examples, scab seems to be a

lot more subtle.

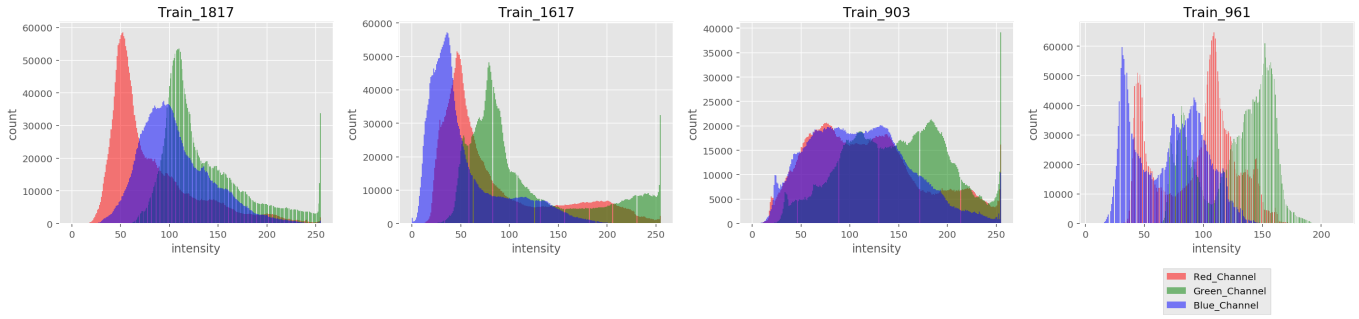
Plants with multiple diseases

Multiple diseases are the most difficult to detect manually for me. In some examples, rust seems quite visible, but other examples look like healthy leafs. It will be interesting, if this class will be easier to detect for the algorithm.

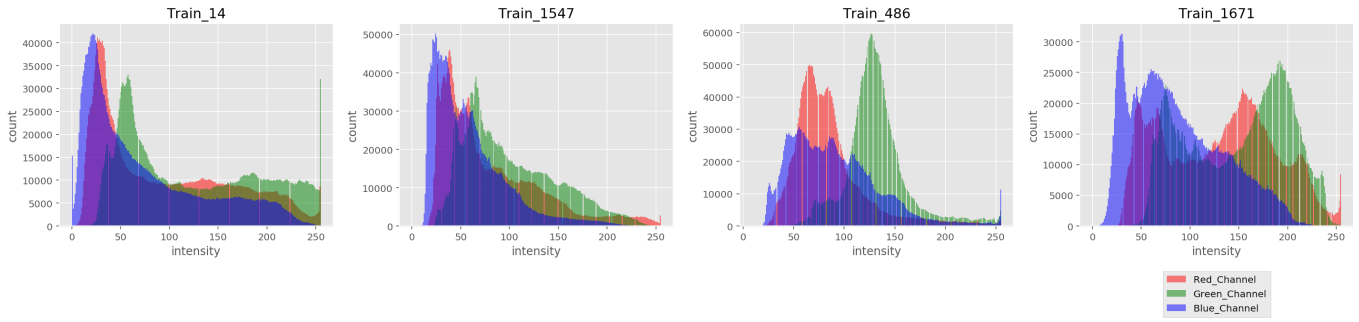
Histogram

In this section, I'm first showing histograms of the color space for some example images. Background of this visualization is, that I want to find out, if the color space can be one feature for the algorithm to classify by. The histograms show the frequency of the intensity for the red, green and blue channel. Looking at single images, the distribution seems to be quite random:

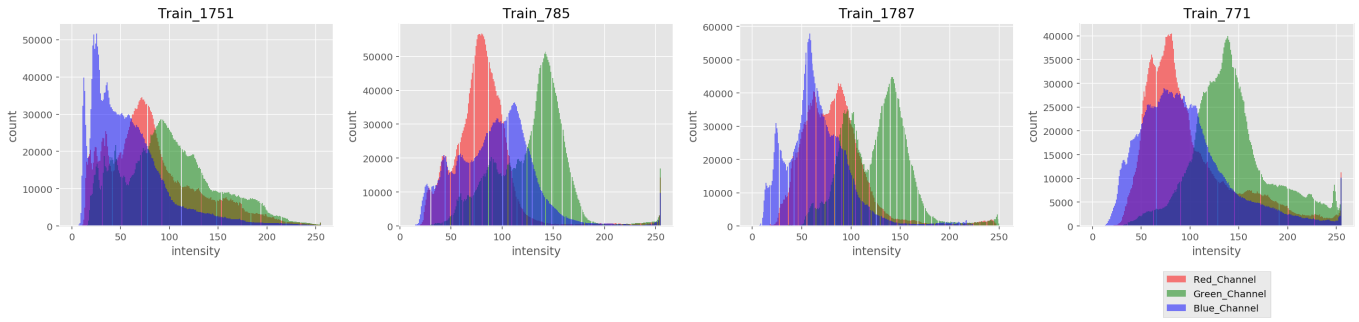
Healthy plants



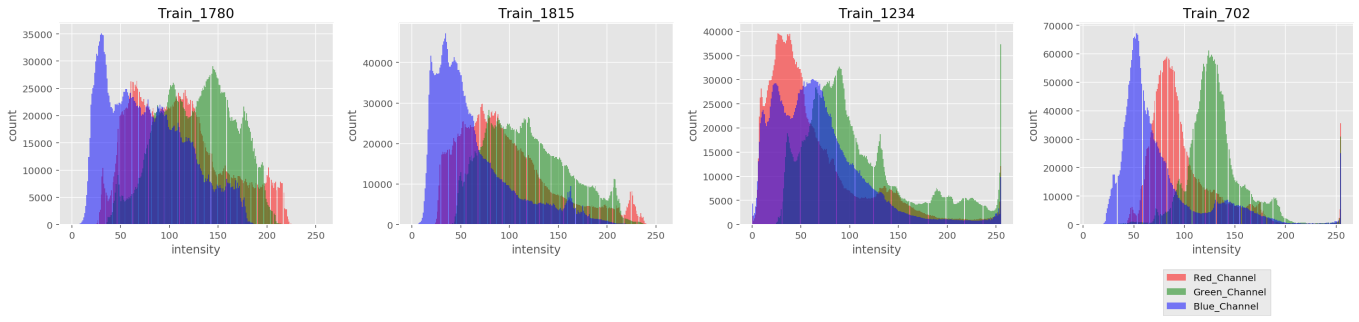
Rust



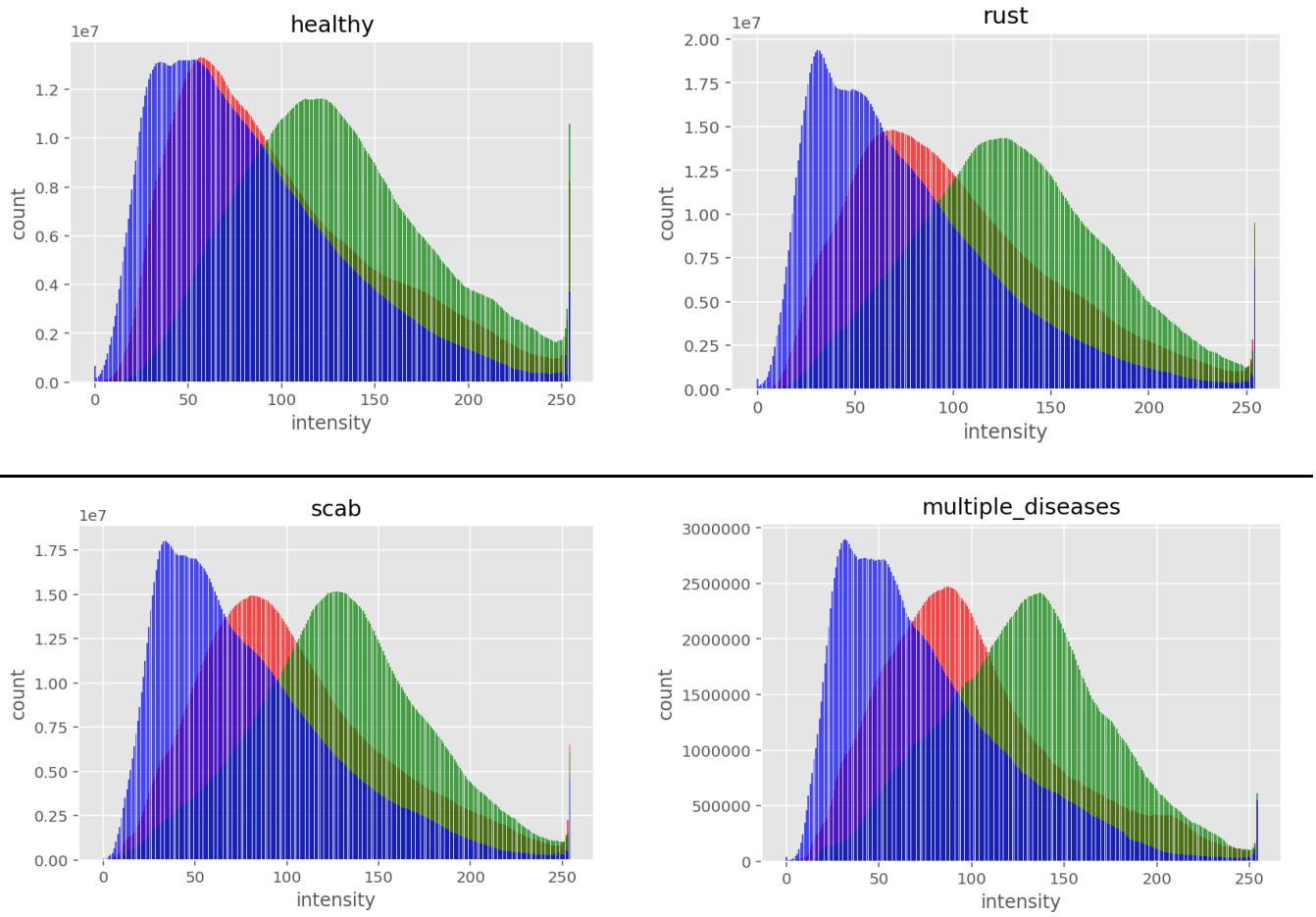
Scab



Multiple dieases



However, in the next visualization I present histograms, that are aggregated over all training examples based on the target class. Here we can see, that the color space of healthy leafs seems to have a spike of intensity for the green channel (this can be seen around value 255 on the x-axis). Scab and multiple dieases show a quite similar color distribution, which might make it hard to distinguish them from each other. Also the distribution of rust looks similar. Interestingly, the shape of the blue channel distribution looks quite similar for rust, scab and multiple dieases.



Inspecting example images and looking at the color space in this chapter, makes me confident that Deep Learnign algorithms are able to perform well on the given task.

Algorithms and Techniques

To tackle the classification task, I chose to try multiple Deep Learning models. I experimented with VGG16, ResNet50, DenseNet121 and MobileNet, but sticked with DenseNet121 (G. Huang, Z. Liu and L. van der Maaten, "Densely Connected Convolutional Networks," 2018.) due to the good performance in early development stages. A common problem of Convolutional Neural Networks is that information can get lost due to increasing depth, both regarding features and the gradient. To increase the information flow, DenseNets make sure that all layers are directly connected with each other and therefore use the information available during training as much as possible throughout the network.

The model will train on images of shape (224,224,3) and use pre-trained weights, that are created during the training on ImageNet. The top layers of the network will be removed and instead I will add the following layers:

- Global Average Pooling
- Batch Normalization
- Dropout with a value of 0.5
- Dense layer with 250 nodes and a ReLu activation function
- Batch Normalization
- Dropout with a value of 0.4
- Dense layer with 4 nodes and Softmax activation function

I choose the optimizer Adam, with the following parameters: learning rate=0.001, beta_1=0.9, beta_2=0.999

All images will be divided by 255. to bring them to a range between 0 to 1. Furthermore, to reduce overfitting, the following data augmentation steps will be applied during training: Rescaling, width shift, height shift, rotation, vertical mirroring and horizontal mirroring.

Benchmark

Eventually, I trained two benchmark models on the available data.

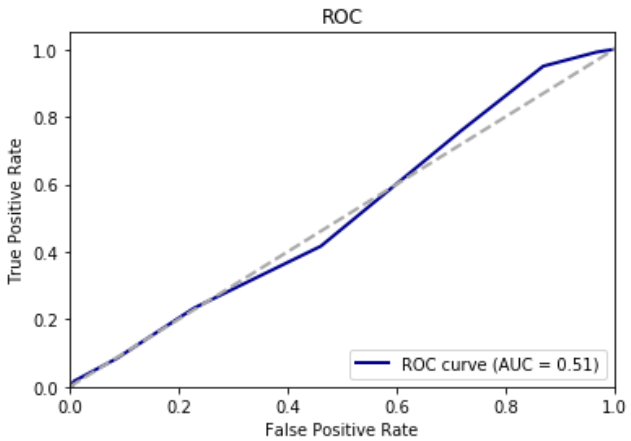
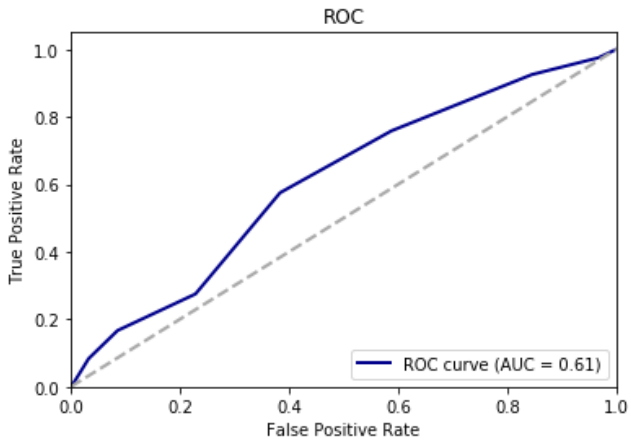
PCA + Random Forest

The first benchmark model extracts 10 patches with the size (25,25) out of each gray-scaled image. Then, PCA is used to learn high-level features present in the data. Those features are then used to train a Random Forest classifier on 80% of the data. On the remaining 20% of the data, the performance of the classifier is evaluated. The performance is extremly poor with an **AUC of 0.51**. One possible factor could be, that I worked with gray scaled images. However, initially I tried this approach with colored images, which also did't lead to better results.

Color histograms + Random Forest

The second benchmark model calculates color histograms for each channel of each image. For a simple aggregations, those histograms use 100 bins instead of the 256 single values. Those features are then used in the very same Random Forest classifier as described above. This leads to an **AUC of 0.61**, which is a slight improvement compared to the results of PCA and Random Forest. With some more feature engineering, this score could probably be improved.

ROC visualizations

Benchmark dictionary learning	Benchmark color hist features
	

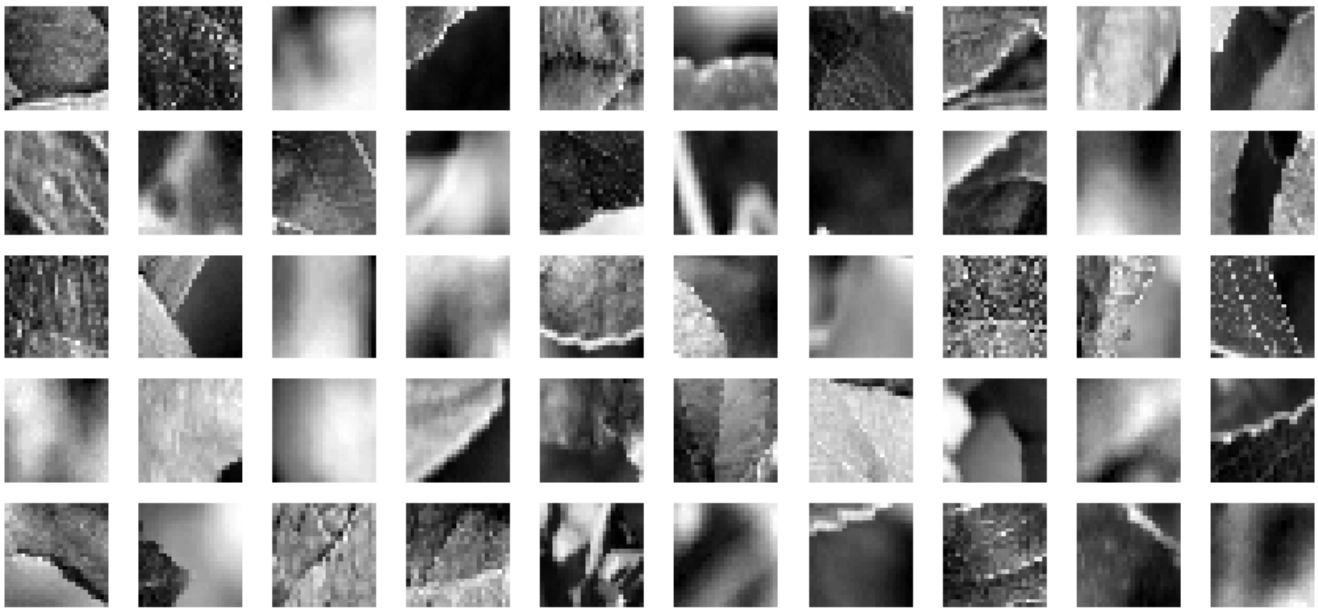
III. Methodology

Data Preprocessing

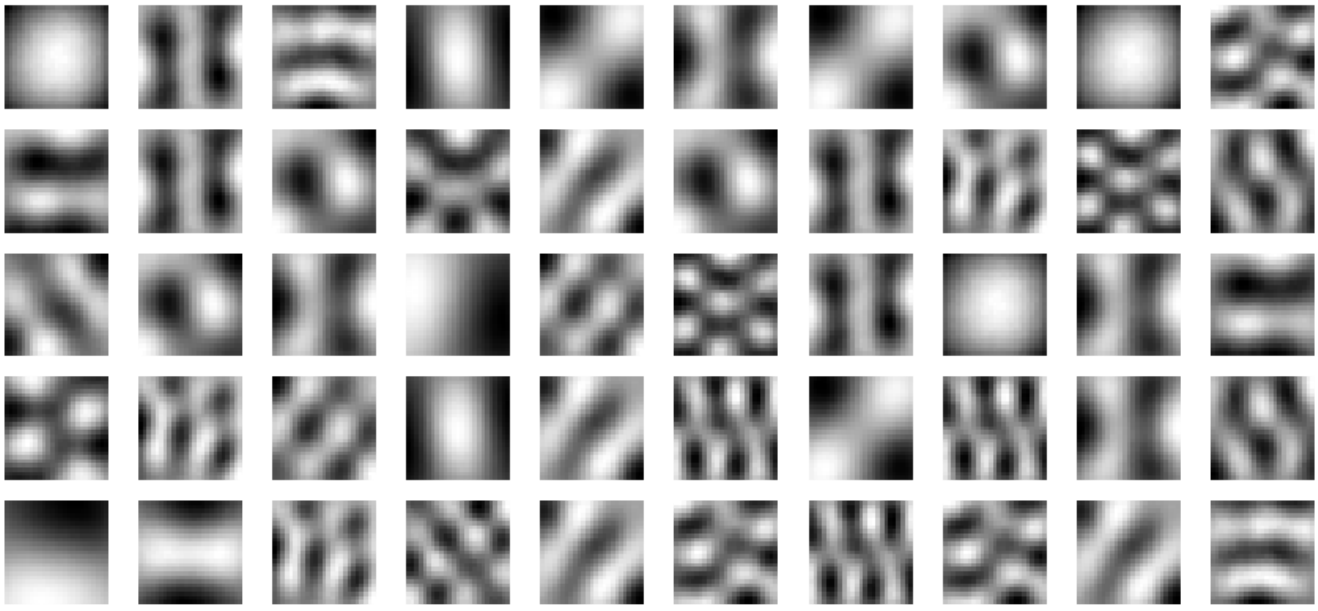
As described in the benchmark section, to classify the data with the Random Forest model, pre-processing was applied. However, the overall results were not satisying, which can be attributed to the pre-processing steps. When training on images, it's quite hard to identify relevant features. The color histograms seem to have some prediction power, but using those alone is not enough for good results.

Here you see two illustrations of the pre-processing of the first benchmark model. The first illustration shows examples of patches that I extracted of each image. The second illustration shows the features produced by applying PCA.

Extracted patches:



PCA features:



This is one of the reasons I'm choosing Deep Learning. Deep Learning takes care of the feature learning step (= pre-processing) automatically. Hence the only pre-processing I'm using is to map the image array into the range (0,1) and applying the following data augmentation steps:

- rescaling to 224,224,3
- width shift
- height shift
- rotation
- vertical mirroring
- horizontal mirroring

First, I applied the data augmentation in real-time during the training by fetching the data from S3 and then performing the augmentation. However, this took a large amount of time for each training epoch. That's why I used the data augmentation only once before training, but ran it three times over the whole training dataset to increase the amount of training images.

Implementation

My implementation is carried out with Keras, which is trained and deployed to Sagemaker. The main model function is called keras_model_fn. I tried different kinds of pre-trained models here, DenseNet121 was initially working the best. I experimented with different amounts of Dense layers and inspected the effect on validation loss. Two Dense layers seemed to work best here.

```
def keras_model_fn():
    model_pretrained = DenseNet121(include_top=False, weights='imagenet',
input_shape=(224,224,3))

    model = Sequential()
    model.add(model_pretrained)
    model.add(GlobalAveragePooling2D())
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
    model.add(Dense(250, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(0.5))
    model.add(Dense(4, activation='softmax'))

    optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999)

    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
metrics=['accuracy'])

    return model
```

Keras generators expect the data to be in a certain structure. We need one folder each for validation and training. Then, each target class image also needs to be stored in a separate folder. I created the following function to achieve this tasks before uploading to S3. I furthermore made sure, that training and validation data will be separated in a stratified fashion to make sure to have an even distribution of class labels.

```
def change_file_structure(df, src_dir="images", tar_dir="train_data",
pred_list=["healthy", "multiple_diseases", "rust", "scab"], val_size=0.2):

    train_dir = os.path.join(tar_dir, "train")
    val_dir = os.path.join(tar_dir, "val")

    for dir_ in [tar_dir, train_dir, val_dir]:
        if not os.path.exists(dir_):
            os.mkdir(dir_)

    for pred in pred_list:
        if not os.path.exists(os.path.join(train_dir, pred)):
            os.mkdir(os.path.join(train_dir, pred))

        if not os.path.exists(os.path.join(val_dir, pred)):
            os.mkdir(os.path.join(val_dir, pred))

    df.index = df.image_id
    del df["image_id"]
    df["pred"] = df[pred_list].idxmax(axis=1)

    preds = df.pred.values
    images = df.index.values

    img_train, img_val, pred_train, pred_val = train_test_split(images,
preds, stratify=preds, test_size=0.2)

    for img_id, img_pred in zip(img_train, pred_train):
        src_path = os.path.join(src_dir, img_id+".jpg")
        tar_path = os.path.join(tar_dir, "train", img_pred, img_id+".jpg")
        copyfile(src_path, tar_path)

    for img_id, img_pred in zip(img_val, pred_val):
        src_path = os.path.join(src_dir, img_id+".jpg")
        tar_path = os.path.join(tar_dir, "val", img_pred, img_id+".jpg")
        copyfile(src_path, tar_path)
```

The following steps in the main function of model.py performs the data augmentation on the data stored in S3 before training the algorithm. As you can see, **STEPS_TRAIN** is multiplied by 3, to increase the amount of training data and augmentation.

```
train_generator = train_input_fn(os.environ.get('SM_CHANNEL_TRAIN'))
val_generator = eval_input_fn(os.environ.get('SM_CHANNEL_VAL'))

STEPS_TRAIN=(train_generator.n//train_generator.batch_size) * 3
STEPS_VAL=val_generator.n//val_generator.batch_size

X_train, y_train = train_generator.next()
for step in range(STEPS_TRAIN-1):
    X_tmp, y_tmp = train_generator.next()
```

```
    X_train = np.vstack([X_train, X_tmp])
    y_train = np.vstack([y_train, y_tmp])

X_val, y_val = val_generator.next()
for step in range(STEPS_VAL-1):
    X_tmp, y_tmp = val_generator.next()
    X_val = np.vstack([X_val, X_tmp])
    y_val = np.vstack([y_val, y_tmp])
```

All the training jobs were running on Sagemaker with GPU instances. The prediction was then carried out by deploying it to a CPU instance, which was enough for inference. Since the inference was triggered out of the notebook with locally stored data, I created a **numpy memmap** to store the pre-processed prediction images, resized to (224,224,3) and value between (0,1). This memmap can be used in your code like a in-memory array, but is actually stored on disk and therefore doesn't allocate too much RAM. It was created as follows:

```
pred_memmap = np.memmap("pred.dat", dtype='float32', mode='w+', shape=
(len(test_imgs),224,224,3))

for i in range(len(test_imgs)):
    img = imread(test_imgs[i])
    img =
np.asarray(Image.fromarray(img).resize((224,224))).reshape(224,224,3)
/255.
    pred_memmap[i] = img
```

The prediction then looked like as follows:

```
predictor = tf_estimator.deploy(initial_instance_count=1,
instance_type='ml.m4.xlarge')

pred_list = []
for idx in range(0, len(pred_memmap), 1):
    prediction = predictor.predict(pred_memmap[idx:idx+1])
    pred_list.append(prediction)
```

Refinement

Multiple deep learning algorithms were tried during the process, as well as data augmentation steps. I evaluated them based based on their local validation score as well as the result on Kaggle, in case they looked good enough to submit. I had ca. 30 different training jobs scheduled on S3 before I started to submit my first solutions. For the Kaggle submissions the data pre-processing steps stayed more or less the same. The amount of epochs changed slightly and was between 25-50. Here you find an overview of the AUROC scores achived in the Kaggle leaderboard:

- VGG-16, pre-trained, only train top layers: 0.582

- VGG-16, pre-trained, only train top layers: 0.498
- DenseNet, pre-trained, only train top layers: 0.706
- DenseNet, pre-trained, train all layers: 0.930
- DenseNet, pre-trained, train all layers: 0.903
- DenseNet, pre-trained, train all layers: 0.950 (include class weights)

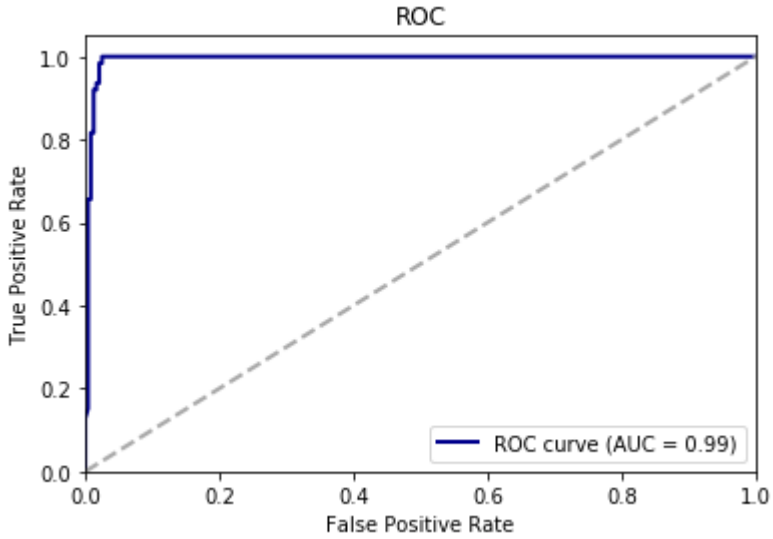
The last solution worked best. Here I put a three-times higher weight on the minority class.

IV. Results

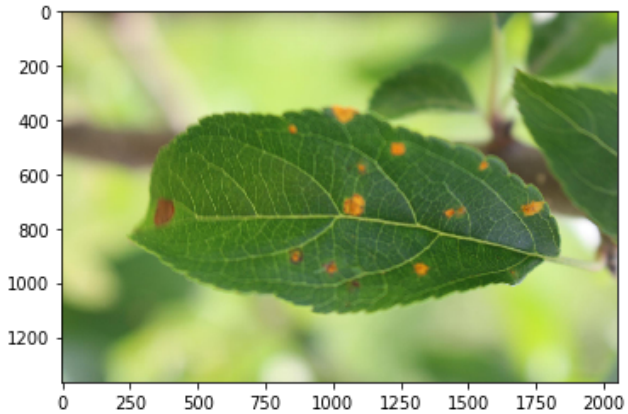
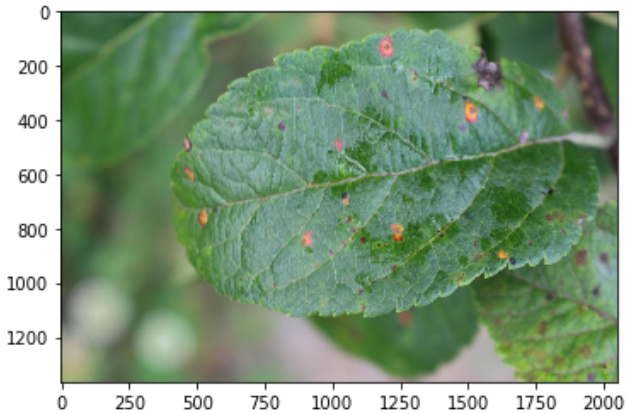
Model Evaluation and Validation

The model was chosen based on the results obtained in the Kaggle leaderboard. As described, before submitting to the Kaggle leaderboard it took multiple iterations to find a feasible architecture where the model loss was small enough to justify a submission to Kaggle.

The final model scores 0.99 AUC on the local validation data and 0.95 AUC on the Kaggle leaderboard, which is a great result to my point of view. The model is able to distinguish between different kinds of plant diseases, but of course still has room for improvement.



To understand the areas, where the model can improve, I looked at the misclassified examples out of the validation data. The main challenge for the model seems to be the class multiple_diseases. This class got misclassified the most as rust. Here I need to inspect, how to further improve the model. One option would be increasing the class weights even further.

Example 1	**Example 2**
	

So while there are still areas for improvement, it's clear that the model generalizes quite well. The difference between validation and test (Kaggle) AUC is reasonable to me and shows, that the validation split chosen is okay. A small deviation is okay and shows, that the validation used by me generalizes well

enough to predict the leaderboard score. With the help of strong data augmentation I tried to make the model unsensitive to data pertubations, which seemed to be successful.

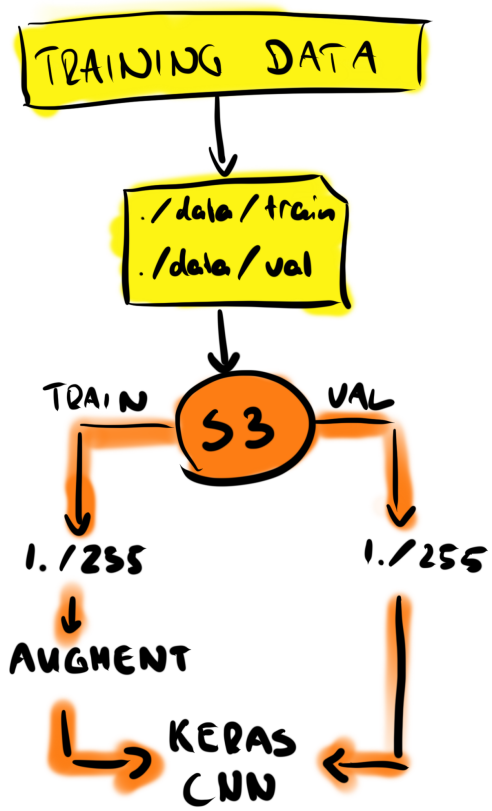
Justification

The results of the model training are significant enough to solve the problem to a reasonable extent. There is still room for improvement, but the model shows definite prediction power. Compared to the benchmark model, the AUC is nearly 0.4 points higher. The ability of Deep Learning to learn features automatically shows clearly how much more well suited those kinds of algorithms are to manual feature learning.

V. Conclusion

Free-Form Visualization

The interesting part for my during the capstone, was the mixture between local data exploration to remote model training/ deployment with Sagemaker, where I needed to store the training data on S3. The combination of "local" and "remote" computing was sometimes challenging, but proved to be very efficient when used correctly. Following I want to give you a very brief summary of the model training in form of a visualization. While the first two steps (marked in yellow) are implemented locally, you can see that the major parts of the project (marked in orange) are implemented with the Sagemaker functionalities.



Reflection

The project was very interesting for me and I was able to apply nearly all of the Nanodegree contents to obtain the final result. I started with simple exploration of the data and then built up the benchmark model. This process was pretty straightforward. The main challenge was training and deploying the Keras model on Sagemaker. It took me many iterations to get a proper training running at all and looking back, I should have sticked to PyTorch. The training and deployment of PyTorch models was described in detail in the course and should save me a lot of time and AWS costs to get a model up and running.

It seems like every Machine Learning framework is handled slightly different in Sagemaker, which makes the transfer of knowledge quite challenging when switching for example from PyTorch to Keras.

However, trying out a different framework was a unique challenge and I learned many new things thanks to it. The main aspects of the project were the following: setting up Sagemaker notebooks, getting the training data from Keras, performing data exploration and preparation in Jupyter, train local benchmark models, setting up a Deep Learning workflow in Sagemaker, transforming and uploading the data to S3 and finally deploying deploying different Deep Learning models to use their results on Kaggle. I'm still far from reaching a top position in this competition, but will try to improve the current model based on my experience in this capstone project.

Improvement

There is definetly a lot of room for improvement for my model. Due to time limitations in this captsone project, I unfortunately can't reflect my future changes here. If I would be more experienced with the automated Hyperparameter Tuning of Sagemaker, I would use it right from the beginning. Trying different Deep Learning models and all their customizations is extremly time consuming, therefore I will use the Hyperparameter Tuning for the on-going development of the model. Using this, I'm confident that the result score should get better without involving too much manual work. I also want to blend the result of different models, because different algorithms might perform better in certain nuances. Stacking the results of each model should therefore help improving the final score.
