AARHUS UNIVERSITY
SCHOOL OF ENGINEERNG

Group 25

Assignment 4
Final project

ITMAL

Name:	Study nr.	Ch. 1	2	3	4	5	6	7	8
Jannik Haumann Lading	201707261	X	X	X	X	X	X	X	X
Alexander Skovby Bjerre	201305878	X	X	X	X	X	X	X	X
Florent Avdijaj	201409781	X	X	X	X	X	X	X	X

Date: 05/05-2020

Table of Contents

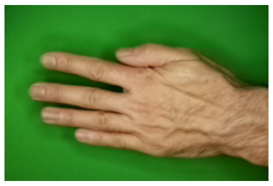
1 Problem

We wanted to make system which could recognize a difference in the hand gestures made in a game of rock, paper, scissors. When playing the game, you can make three different shapes with your hand. If we are able to make a system that could recognize the difference in a the hand gestures, then the game could be played in different ways, eg. over the computer.
This assignment will describe an end to end machine learning process, getting the data ready, picking, training and optimizing an algorithm to recognise images rock, paper, scissor hand gestures.

2 Dataset

The dataset which we are going to work on, will be a dataset taken from kaggle. It is a dataset of 2188 pictures, almost equally distributed between pictures of hands making rock, paper and scissor hand gestures. All images are RGB images of 300 pixels wide by 200 pixels high in .png format. They are all taken on top of a green background made out of a piece of cloth. The picture are taken with different hands, and some are right handed and some are left handed. Most of the hand gestures comes in from the right border of the image, but all of the hand gestures are not made in the same way, and the hands are not placed exactly in the same place. Examples of issues with the data is hands being too close to the camera filling the screen, coming into the picture from weird angles, or the same gesture done in different ways, such as paper with finger sticking together, and paper where there's a lot of space between the fingers.

(a) Example rock


(b) Example paper

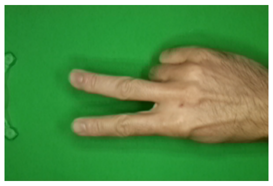

(c) Example scissor

Figure 1: Three examples of pictures from dataset

2.1 Analysis of the dataset

In order to gain some insights about the dataset, an analysis of the data set is to be done.

2.1.1 3 most principal components analysis

The group has chosen to do a 3 most principal components analysis of the dataset, in order to see if there's any obvious patterns in the dataset that emerges.
In the 3 most principal component analysis, the group reduces the dataset to the 3 most principal components, and then create scatter plots where the different types of pictures are marked differently. This allows the group too see if there's any immeadiate patterns that show up, which implies a relationship between the data classification and the values of the 3 most principal components of the data.
If a pattern emerges in this analysis, then it is a good indicator that the group should be able to create an algorithm, which is able to classify the dataset.

The analysis has been done on both the pristine dataset, and the cleaned up dataset. The result is the following 3 pairs of 3d plots:

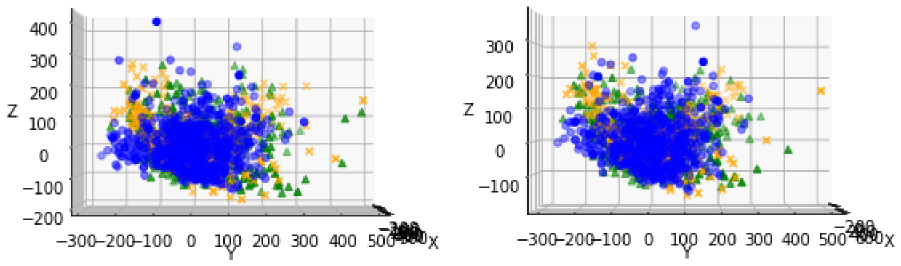


Figure 2: Pristine dataset on the left, cleaned up on the right

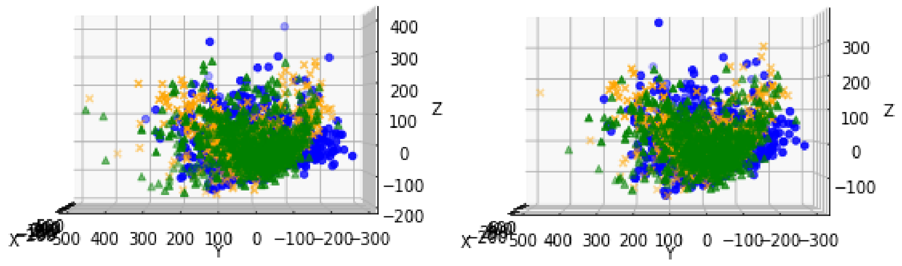


Figure 3: Pristine dataset on the left, cleaned up on the right

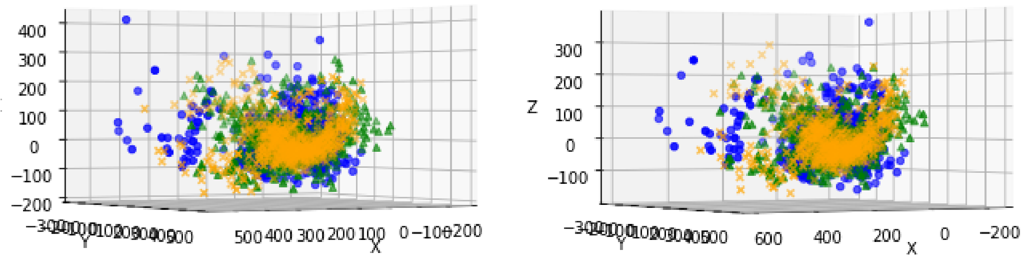


Figure 4: Pristine dataset on the left, cleaned up on the right

From doing the analysis we gain two insights about our data.
The most important insight is that a pattern emerges between the classification of the data, and the values of its' 3 most principal components. While it is hard to distinguish between the different shape patterns, as they are all mashed a bit together, we find that by rotating the angle of the plot, different shapes comes to the foreground of the plot. This tells us that the different shapes have values where they tend to be in the plot. This in turn means that the group should be able to make an algorithme, that can predict the the hand sign of an image.

The group also finds that looking at the plots it becomes clear, that cleaning up the dataset, has resulted in less outlier data points.

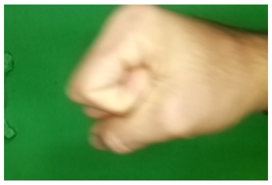
3 Preprocessing

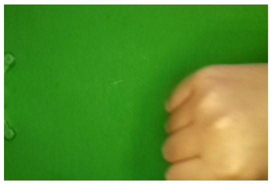
Preprocessing is about making the data ready, for the actual training. Removing unnessary data, polishing the data to represent the data in the best possible way, and in the most efficient way.

3.1 Removing bad data

For a starter we took our dataset and removed some of the bad pictures. Some of the pictures we thought to be to odd, for the general picture of how the three hand gestures should be.

On figure 5 three pictures are displayed, all pictures bad in their own way. But these are just some examples of what we removed. Similar bad pictures where found for the scissor and paper hand gesture.


(a) Blurry and different positioning


(b) Wrong placement

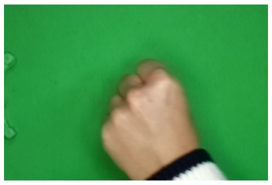

(c) Wrong placement

Figure 5: Three examples of pictures that was removed

3.2 Removal of background and greyscale transform

Because the pictures in the dataset was taken with a green screen behind it, one thing that could be done was removing the background, to hopefully reduce noise from the green, we also removed the colors, so the pictures where made into a grayscale, the code can be seen in listing 1.

```python
import cv2
def removeGreenScreen(image_path):
    img = cv2.imread(image_path)

    image_copy = np.copy(img)

    lower_green = np.array([0, 50, 0])      ##[R value, G value, B value]
    upper_green = np.array([100, 255, 100])

    mask = cv2.inRange(image_copy, lower_green, upper_green)

    masked_image = np.copy(image_copy)
    masked_image[mask != 0] = [0, 0, 0]
    masked_image = cv2.cvtColor(masked_image, cv2.COLOR_BGR2RGB)
    return masked_image

# reshape images to not include the rgb parameter (3)
def reshapeImgs(images):
    newImages = []
    for image in images:
        # Remove rgb dimension
        img = image[:, :, 0]
        # 200, 300 to 60.000 pixels
        img = np.ravel(img)
        newImages.append(img)

    return newImages
```

Listing 1: Removing green screen and making pictures grayscale

The result of this preprocessing step can be seen on figure 6.

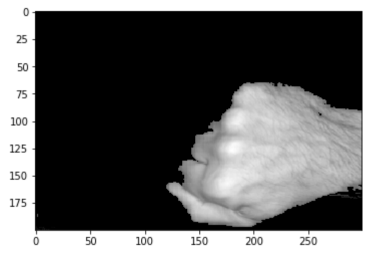


Figure 6: Background is blacked and picture is gray

3.3 PCA reduction

PCA reduction is a technique used for reducing overfitting. Overfitting is usually the result of having too many features/components for each data point, as it is then likely that a lot of the features aren't very useful for predicting the data entry. At the same time when the amount of features increases, the amount of training data needed increases exponentially, in order to fill up the capacity space and avoid overfitting.

Since our dataset consists of images, there's bound to be a lot of not very useful features. An example of why is because our images are mainly centered with hands coming in from the right side. The result of this is that the top, left and bottom border areas of the images in general won't tell anything about the class of the image, as they are always green no matter the class. As such removing these pixels/features from the data, won't cause much noticeable loss in the results of a machine learning algorithme.

In order to remove these bad features we have used Principal Component Reduction. The first step is to find out how many components it will make sense to reduce down to. In order to do this we create a graph showing the cumulative sum of explained variance of the components.

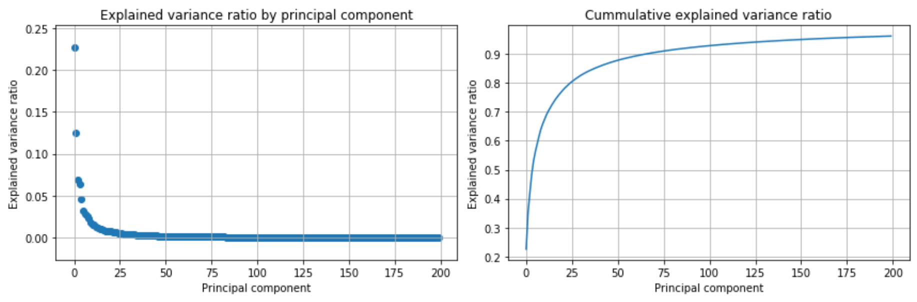


Figure 7: Left: Explained variance by component. Right: Cumulative sum of explained variance of components

From the above figure 7 we find that if we reduce down to the 200 most important components, then these 200 components will still explain around 95 per cent of the variance.

Based on this information, we decide to add PCA reduction, with the 200 most principal components, to our preprocessing pipeline, as seen in listing 2

```
pca = PCA(n_components=200)
reshaped_imgs_pca = pca.fit(reshaped_imgs)
imgs_pca = pca.transform(reshaped_imgs)
```

Listing 2: PCA reduction

The result of this is less overfitting and faster training.

4 Algorithms

To find out what algorithms were suitable for our dataset, we used the cheatsheet introduced to us in class for guidance, as seen in figure 8. Since we want the model to predict one of three categories, and our data is labeled, we believe that we should use a classification algorithm. Since our dataset is very small, we decide to try some of the algorithms suggested by the cheatsheet for datasets below 100k data samples. We decide to choose three candidate algorithms that seem suitable for the dataset; Linear SVC, KNeighbors Classifier and SVC.

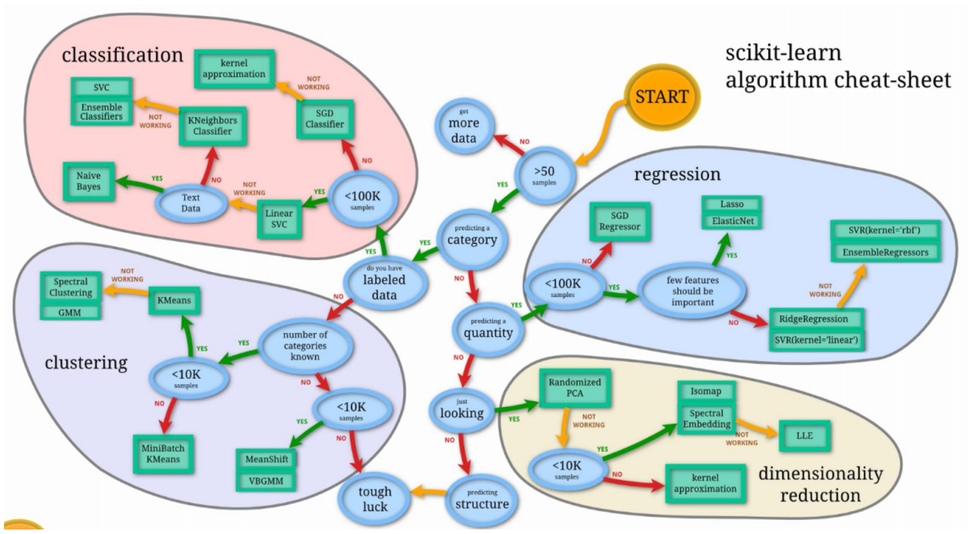


Figure 8: Cheat sheet for picking ML algorithm

Starting out we take a quick look at these algorithms, to get an impression of how well they fit to the data. As such we try fitting each of the algorithms, with just one parameter configuration, as seen in listing 3

```
linearSVC = LinearSVC(random_state=0, max_iter=10000)
linearSVC.fit(X_imgs_training, y_training)
linsvc_y_pred = linearSVC.predict(X_imgs_test)
print(f"LinearSVC F1 Score: {f1_score(y_test, linsvc_y_pred, average='micro')}")

neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X_imgs_training, y_training)
neigh_y_pred = neigh.predict(X_imgs_test)
print(f"KNeighbors F1 Score: {f1_score(y_test, neigh_y_pred , average='micro')}")

svc = SVC(gamma='auto')
svc.fit(X_imgs_training, y_training)
svc_y_pred = svc.predict(X_imgs_test)
print(f"SVC F1 Score: {f1_score(y_test, svc_y_pred, average='micro')}")
```

Listing 3: Quick test of the candidate algorithms

We find the following F1 scores:

Algorithm	F1 Score
LinearSVC	0.78
KNeighbors	0.968
SVC	0.828

We quickly see that of the three candidate algorithms, k-neighbors seem to be the best candidate. However due to the fact that we are working with a small dataset, training a model takes very little time. Therefore we believe that instead of choosing k-neighbors right away, we should use the short training time, keep all of the candidate algorithms for now, and take the final decision after having found and tried fitting with the best hyper-parameters for each of the algorithms.

4.1 KNeighbors Classifier

The KNeighbors Classifier is a classification algorithm, that decides the classification by looking at the data which is most similar, and by specifying how many "neighbors" you want your classification to rely on, it will take the classification that is represented the most, in the "neighbors". So depending on how the data is, you have to pick the neighbor count accordingly. You can also tweak the weight of the neighbors by applying some kind of scaling for how far away the neighbors are. By this the nearest neighbors will have the most influence. This is an algorithm which is very vulnerable when it comes to outliers, because outliers will probably not be with the group of data, so the chance of it having neighbors not like itself is high.

4.2 SVC

SVC is a learning algorithm that is a part of the supervised learning SVM (supprt-vector machines) methods. They are typically used for classification which is relevant for this project, as the pictures are classified into the 3 categories rock, paper and scissors.
When classifying the data, there is a margin to each classification for the data to fall under. The margins can be manipulated with, and can be changed so that misclassification happens less frequently.

SVC stands for C-Support Vector Classification. The C parameter for the SVC algorithm is the regularization parameter. It determines the influence that the L2 penalty has when classifying the data, which means the higher the value of C is, the less chance there is of misclassifying the data, where the margin between the different classifications also become smaller as a result.
For the SVC algorithm, different parameters can be applied, but in this case, only what will be deemed as relevant parameters will be experimented with.

4.3 Linear SVC

Linear SVC is the same algorithme as SVC, but where the kernel is set to linear. A problem with SVC is that the fit training time scales quadratically with each data sample, meaning that training time quickly becomes long as the dataset grows. Therefor Linear SVC can be a better choice for big datasets.
However our dataset is a small dataset, and as such this is not really useful to our project. What however could be useful, is the fact that linear SVC allows for more experimentation, as it has hyper-parameters for loss and penalty functions, which might result in a better algorithm.

5 Optimizing

Optimization is about taking the algorithms we have decided on, and tweaking them to be optimized for our specific dataset.

5.1 Performance metrics

When trying to optimize an algorithm for our dataset, we need a way to compare how the differently tweaked versions of the algorithms perform, and as such we need to decide on a performance metric.

In the context of our machine learning project, predicting hand gestures, we can't really argue that having a high precision, and few false positives, is more important than having a high recall, and having few false negatives.
The f1_score combines the precision and recall scores, such that their results are weighted equally in the final score, according to the formula:

$$2pr/p + r$$

Due to these considerations the group decides to use the f1_score as the performance metric to compare algorithms.

5.2 Hyper-parameters

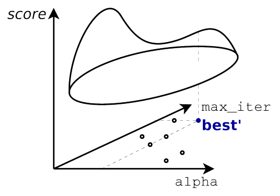
The way we try to optimize our algorithms, is by testing them on the dataset with different hyper-parameters.
In the following listing 4, we defined the hyper-parameters that we wanted to test combinations of.

```

linearsvc_tuning_parameters = {
    "penalty": ('l1','l2'),
    "loss": ("hinge", "squared_hinge"),
    "dual": [False, True],
    "tol": [1e-3, 1e-2, 1e-4],
    "C": [0.2, 0.5, 1 ],
    "multi_class": ("ovr", "crammer_singer"),
    "fit_intercept": [False, True],
    "intercept_scaling": [0.2, 0.5, 1, 1.5],
    "max_iter": [500, 1000, 2000, 5000]
}

kn_tuning_parameters = {
    'algorithm': ('auto', 'ball_tree', 'kd_tree'),
    'leaf_size': [10, 20, 30, 50],
    'n_neighbors': [1, 2, 3, 5, 7],
    'weights': ('uniform', 'distance'),
    'p': [1, 2]
}

svc_tuning_parameters = {
    'kernel': ['rbf', 'linear', 'poly', 'sigmoid'],
    'degree' : [1, 3, 5, 10],
    'gamma' : [1e-3, 1e-4, 1e-5],
    'C' : [1, 10, 100, 1000, 1000],
    'max_iter' : [10, 100, 1000, 10000],
}
```

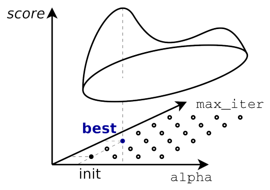
Listing 4: Hyper parameters for tuning of models

5.3 RandomizedSearchCV and GridSearchCV

To optimize our algorithms for the dataset, we made the decision of using randomized search, and supplementing with grid search.
First we tried using RandomSearchCV. The reason was that if the dataset took a long time to run, then this method had a clear advantage by maybe reaching a good solution in a shorter amount of time. This was definitely a thing we wanted to try.



(a) Concept of randomsearch



(b) Concept of gridsearch

Figure 9: Tuning of models

RandomSearchCV works by feeding it a selection of hyper-parameters, and other parameters such as cross-validation folds, performance metric to use, and number of iterations. When it is run, it will try random combinations of the the hyper-parameters until it has done so for the specified amount of iterations, and return the best combination found. This is visualized on figure 9a.

The LinearSVC model has 4608 possible combinations given the chosen tuning hyper parameters. This meant that a substantial amount of time could be saved if we tried a random search with 20-100 iterations. The KNeighborsClassifier only had 240 combinations and SVC had 960, so these were not as affected by the lesser amount on runs. But the effect of the random search would still be pretty good.

Each iteration of the RandomSearchCV were quite fast. Because the dataset is relatively small, it was decided that GridSearchCV should also be tried, as we expected it to be be fast as well, and it might find a better result. The GridSearchCV as seen on figure 9b, is systematically going trough the each combination, and finding the best one.

6 Pipeline

The pipeline code for this project runs through all the steps to reach the final model(s). It is run sequentially and each steps output feeds the next steps input, hence the name pipeline. One thing to note is that the pipeline can be divided into many steps but it can be divided into 3 essential steps preprocessing, data-splitting and model selection.

6.1 Preprocessing

In the first step of the preprocessing stage the images are loaded, and afterwards the loaded data is manipulated with so that the images no longer will have the color attributes and the dimensionality of the pictures are transformed from having 200x300 pixels dimenstionality into a 1D array of 60.000 features, representing all the pixels for the image. The reason as to why was these decisions came to be, can be seen in section 3. To be able to perform PCA on the dataset the pixels had to be changed to a 1D array.

```python
# Function for loading the images
def loadImages(path):
    imagesList = os.listdir(path)
    loadedImages = []
    for image in imagesList:
        img = imread(path + image)
        loadedImages.append(img)

    return loadedImages

rock_imgs = loadImages(path_rock)
paper_imgs = loadImages(path_paper)
scissors_imgs = loadImages(path_scissors)

imgs = rock_imgs + paper_imgs + scissors_imgs
```

Listing 5: Pipeline code. Loading images

Depending on where the images are located on the computer, the path is set to the location and then the images are loaded. The loaded images are then processed so that the color/RGB parameter is set to null as seen in section 3 and listing 1 to transform the loaded images into black and white images.
The data had to be labeled as well which was done by labelling as such, rock = 0, paper = 1 and scissors = 2.

```python
# y array to define the different options
# rock = 0, paper = 1, scissors = 2
def generateYArray():
    # Rock
    all_imgs = np.zeros(len(imgs))
    # paper
    all_imgs[len(rock_imgs):len(rock_imgs) + len(paper_imgs)] = 1
    # scissors
    all_imgs[len(rock_imgs) + len(paper_imgs):] = 2

    all_imgs = list(all_imgs)

    return all_imgs

y = generateYArray()
```

Listing 6: Pipeline code. Generation of labelling data

and after this the data is ready to be exposed to PCA (principal component analysis), for further preprocessing. To reduce the features of the datasets PCA and reduction dimenstionality was used. To see detailed explanations on how PCA was used on this project refer to section 3.3 and listing 2 for the code section that is responsible for the PCA reduction.

6.2 Data-splitting

As is always the case when testing and verifying a model, the data should be divided to training and test sets. The training sets are there to work on them and test out different models before deciding to move on to the test set and verify that the model holds true,

It was important to have a way of splitting and shuffling the data so that both the image- and label data was still going to be in the same order, since the labelled data is the indicator of what the image is showing.

```python
# Copy the datasets, otherwise the actual datasets are going to be shuffled
shuffled_imgs = imgs_pca.copy()
shuffled_y = y.copy()

# Zip the datasets to be able to shuffle them in the same order
c = list(zip(shuffled_imgs, shuffled_y))

# Shuffle the datasets, so that they match in order
random.shuffle(c)

# Get the shuffled lists
shuffled_imgs, shuffled_y = zip(*c)
shuffled_imgs = list(shuffled_imgs)
shuffled_y = list(shuffled_y)

# Split data into training- and test sets

# Training
X_imgs_training = shuffled_imgs[500:]
y_training = shuffled_y[500:]

# Test
X_imgs_test = shuffled_imgs[:500]
y_test = shuffled_y[:500]
```

Listing 7: Pipeline code. Shuffling and splitting the images data

The limit on where the training and image data split was set to 500 since there were 2072 images left when the bad images had been discarded, where the ratio between the training- and test data is set to roughly 25%. After the data is split into training- and test sets, the pipeline proceeds to the model selection section.

6.3 Model selection

After the data is preprocessed and split, the search for the best model and hyperparameter tuning now take place in the last part of the code. To start off, as explained in section 4 and shown in listing 3, the algorithms suggested in the cheat sheet (figure 8) were tested for if they would be a good solution for this project.

7 Results

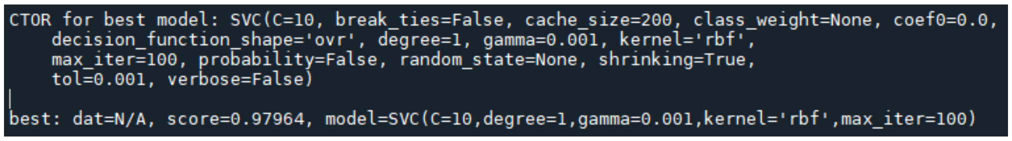
In this section we will present the the final results for each of the three algorithms, based on results of GridSearchCV and RandomSearchCV

For all three algorithms, the model selection algorithms RandomizedSearchCV and GridSearchCV were performed to find the best model with optimal hyper-parameters.

We have also tried to find the best hyper-parameters for when the preprocessing includes greyscaling and removing the green background, in order to conclude if this preprocess increases the score of our algorithms.
To get a good visual representation of the results from the model-searching process, some code functionality was copied from the course "MAL" in lesson 9 and used in this project, to get a clear systematic representation of the results. Some of the output can be seen on all the screenshots provided in this section.

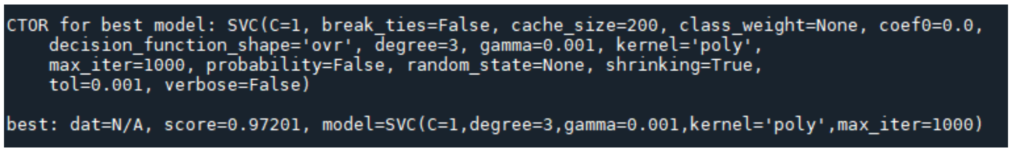
7.1 SVC

The following is the results for the SVC algorithm.



```
CTOR for best model: SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=1, gamma=0.001, kernel='rbf',
    max_iter=100, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

best: dat=N/A, score=0.97964, model=SVC(C=10,degree=1,gamma=0.001,kernel='rbf',max_iter=100)
```

Figure 10: SVC best model results, green screen included



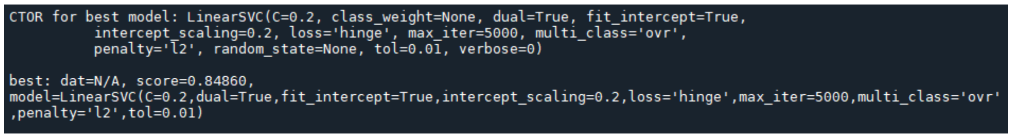
```
CTOR for best model: SVC(C=1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.001, kernel='poly',
    max_iter=1000, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

best: dat=N/A, score=0.97201, model=SVC(C=1,degree=3,gamma=0.001,kernel='poly',max_iter=1000)
```

Figure 11: SVC best model results, green screen removed

Figures 10 and 11 show the constructor and the hyper-parameters for the best model that was found.
The best algorithm is found when the green screen is included. The best score is 0.9796 out of 1.0. The score is 0.9720 when them images are greyscaled and the green background removed.
As such for the SVC algorithme, greyscaling reduces the f1_score by 0.0076.

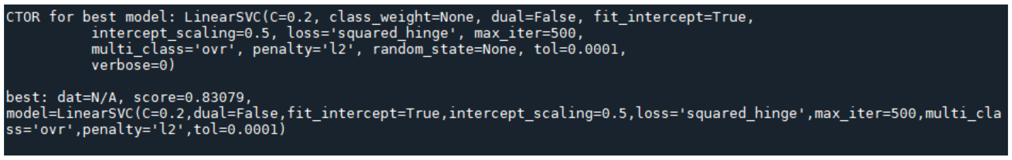
7.2 LinearSVC

The following is the results for the LinearSVC algorithm.



```
CTOR for best model: LinearSVC(C=0.2, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=0.2, loss='hinge', max_iter=5000, multi_class='ovr',
          penalty='l2', random_state=None, tol=0.01, verbose=0)

best: dat=N/A, score=0.84860,
model=LinearSVC(C=0.2,dual=True,fit_intercept=True,intercept_scaling=0.2,loss='hinge',max_iter=5000,multi_class='ovr'
,penalty='l2',tol=0.01)
```

Figure 12: LinearSVC best model results, green screen included



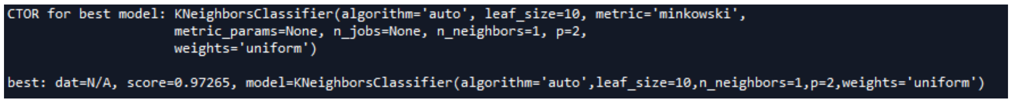
```
CTOR for best model: LinearSVC(C=0.2, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=0.5, loss='squared_hinge', max_iter=500,
          multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
          verbose=0)

best: dat=N/A, score=0.83079,
model=LinearSVC(C=0.2,dual=False,fit_intercept=True,intercept_scaling=0.5,loss='squared_hinge',max_iter=500,multi_cla
ss='ovr',penalty='l2',tol=0.0001)
```

Figure 13: LinearSVC best model results, green screen removed

Figures 12 and 13 shows the constructor and the hyper-parameters for the best Linear SVC model that was found. The best model found is on the images where the preprocess doesn't include removing the green screen. The best score is 0.8486 out of 1.0 The score is 0.8486 when the greenscreen background is removed.
This means that for the LinearSVC algorithm, removing the greenscreen decreases the f1_score by 0.01781.

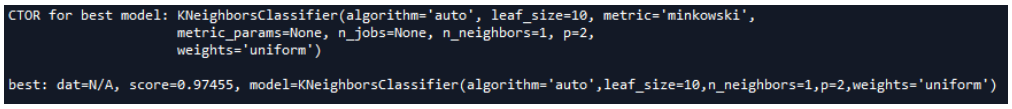
7.3 KNeighbors Classifier

The following is the results for the KNeighbors algorithm.



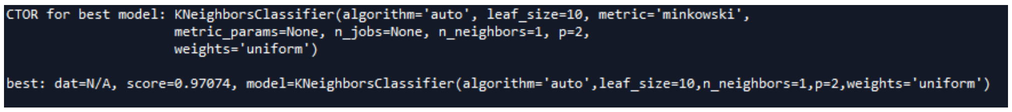
```
CTOR for best model: KNeighborsClassifier(algorithm='auto', leaf_size=10, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=1, p=2,
                     weights='uniform')

best: dat=N/A, score=0.97265, model=KNeighborsClassifier(algorithm='auto',leaf_size=10,n_neighbors=1,p=2,weights='uniform')
```

Figure 14: KNeighbors classifier best model results, green screen included



```
CTOR for best model: KNeighborsClassifier(algorithm='auto', leaf_size=10, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=1, p=2,
                     weights='uniform')

best: dat=N/A, score=0.97455, model=KNeighborsClassifier(algorithm='auto',leaf_size=10,n_neighbors=1,p=2,weights='uniform')
```

Figure 15: KNeighbors classifier best model results, green screen removed



```
CTOR for best model: KNeighborsClassifier(algorithm='auto', leaf_size=10, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=1, p=2,
                     weights='uniform')

best: dat=N/A, score=0.97074, model=KNeighborsClassifier(algorithm='auto',leaf_size=10,n_neighbors=1,p=2,weights='uniform')
```

Figure 16: KNeighbors classifier best model results, green screen included of the full dataset. No PCA.

Figures 14, 15 and 16 shows the constructor and the hyper-parameters for the best model that was found. The best model found is on the greyscaled images without the green screen. The best score is 0.9746 out of 1.0 The score is 0.9727 when the green background is not removed. And the score is 0.97074 when the dataset is not tampered with except reshaping it.
This means that for the KNeighbors algorithm, removing the greenscreen increases the f1_score by 0.0019.

8 Discussion and conclusion

In this project the objective was to find a way to predict whether an image is showing rock, paper or scissors from a set of images. To make it more manageable to find a model that fits the objective of this project, a number of steps were performed on the dataset to reduce the complexity of the dataset, resulting in a compressed dataset consisting of only fundemental parts needed for this project, as well as cutting down on time spent for operations. This is called the preprocessing step.

To be able to predict what the images are, a good model had to be found. Searching for a model consisted of testing the model on the dataset and comparing the results. The testing was done with different variations of both the dataset and models used. The models were tested on two versions of the dataset, one were the greenscreen on images were included, and one where the greenscreen was removed in the preprocessing stage. The models were also tested on a dataset where PCA reduction was applied, and one where PCA reduction was not applied.
In the beginning the chosen models were tested on the dataset, by performing a quick test with minimal tweaking of the hyperparameters. Later on the hyperparameters were tweaked to find the best model. When tweaking with the hyperparameters, RandomizedSearchCV and GridSearchCV were used to find the best model with optimal hyperparameters.

For the KNeighbors classifier it is seen in section 7.3 that the best score value is found when the green screen is excluded from the pictures. As the name implies, KNeighbors, the algorithm compares the features/data entries to the n-amount nearest neighbors, and therefore it makes sense that in this case the score value is higher when the green screen is set to one color (black), because it is easier for the algorithm to differentiate between what is part of the hand, and what the rest of the image is.

For the SVC algorithm, one detail between the best model for the dataset with and without the greenscreen, is that when operating on the images without the greenscreen, the best model is one with the hyperparameter "kernel" set to "poly" (see figure 11 which stands for polynomial.) The reason as to why polynomial is chosen to be the optimal hyper-parameter for "kernel", can be because of the increased numbers of the same outlier features on the screen, because the whole background is changed to black, and a lot of the features have the same value. The polynomial kernel include more features the higher the amount of the degrees is, with the best model found to be with 3 degrees.

In section 7, the results show that the best model is found to be the SVC model, used on the dataset where the green screen is not removed from the images. The score has a value of 0.9796 out of 1.0, which means the model is very likely to predict if it's going to be a rock, paper or scissors image. The scoring is not that far ahead of the KNeighbors model, which scored 0.9746. The SVC model is however much better than the LinearSVC model, which only scored 0.8486.

In regards to why the LinearSVC is scoring so far behind the other two models, we believe it might be because LinearSVC is meant for use on bigger datasets, and ours is quite small.

The cheat sheet shown in figure 8 has proven to be very useful, since all models yielded good results and all are applicable for this project.

Tweaking the hyper parameters gives us some interesting results. When doing it on KNeighbors the model is less than one procent better. But this is also because the default parameters just fit so well. But when it is done on the SVC the result is made extremely better, and it actually surpasses the best result of KNeighbors. The overall improvement of how precise the best model was has not changes a lot, but gaining that one percent could in some cases be a very good thing, and by doing this, we found out that the best model was not the one we first assumed.
Dealing with hyperparameter tuning and finding the best model for the dataset, can take a long time. The trade-off between time and results in this case is that it takes a lot of time to find the best model but the result is only a slight improvement.

There can be a lot of reasons as to why the difference in scores is so small, since one would expect it to be bigger. The size of the dataset is first of all not that large, but images can still be hard to deal with despite the size of the dataset, since images can have a lot of features. The dataset is not a bad dataset, since it consists of images that are "prepared", meaning the only thing on the image is the hand, a green screen and the only "bad" data that can be is where there may be some clothes on the arm.
The dataset therefore does not contain far off outliers, and some pictures that were deemed as outlier pictures from the project group were removed from the dataset.
The RGB parameter was removed from the data to turn them into grayscale pictures to decrease the dimensionality of the data.
The data was further decreased in complexity when performing PCA on the dataset, reducing the amount of features from 60.000 (200 x 300 pixels) to a total of 200, which is a significant decrease in features and complexity. So before even searching for a model to fit the data, the data has been preproccessed to a degree where any amount of outliers is at a minimum, and therefore when searching for a model to fit the data, tweaking the hyperparameters will not yield a big result.

As a last test, the dataset went through the pipeline one more time, but this time without performing PCA on the dataset, to see it there were any significant variation from the PCA reduced dataset. This was only tested for the KNeighbors classifier model, see figure 16. As seen from the scores in section 7.3 the score difference is very minimal. This means that performing PCA on the datasets can be a viable method and results in a more manageable dataset to train on. One of the most measurable reasons to not use the dataset with no PCA reduction is the factor of time, time spent training was considerably larger.