Signature verification

Alpen-Adria-Universität Klagenfurt

Philipp Freislich, Muamer Hrncic, Daniel Siebenhofer

February 2022

Contents

1 Abstract

Confirming the authenticity of a handwritten signature is of great importance in many cases. A major hurdle is the often great similarity between forged and genuine signatures. In this project, we address an online signature verification problem, in which handwritten signatures using a tablet are compared to reference signatures. Our proposed method of achieving this task consists of three main steps, preprocessing, feature extraction and classification using support vector machines (SVMs). We evaluate different preprocessing setups in terms of their influence on the classification of the given signatures. The experiments in this project are performed on our own dataset, which consists of the signatures of the members of our team. This project highlights the difficulties of validating the authenticity of handwritten signatures and the influence of different preprocessing setups on this process.

2 Introduction

Handwritten signatures are one of the oldest and most widely used biometric authentication techniques in administrative and financial institutions due to its simplicity and uniqueness [1]. As technology progresses, authentication methods have also evolved. Handwritten signatures are now categorized as online signatures and offline signatures. Online signatures have much more distinct features than offline signatures; therefore, they are easier to verify [2]. Another advantage of using online signatures is that there is close to no noise present in the input image, which makes preprocessing the image easier and cheaper. Nowadays capturing an online signature is no longer considered as expensive as in earlier times, primarily due to low cost signature pads and tablets being widely available and used. The dataset used in this project contains genuine signatures of every member of our team with random and skilled forgeries created by ourselves, trying to imitate the genuine signature. As one tends to sign signatures in a quite similar manner when they are signed one after another, we collected the signatures in multiple sessions, to prevent this from happening. In real-life applications, genuine signatures of a person vary considerably, because it is very unlikely to sign exactly the same way every time one signs a document. To capture as much intra-class variety of our signatures as possible, we prepared our signatures for the dataset over a time span of two weeks. For this project we focus on online signatures, due to the viability reason mentioned above and because of the shift of nearly every possible aspect of life from offline to online, so we predict that online signature verification will be the more future oriented approach to signature verification. The verification of a signature plays a vital role in administrative and financial settings and therefore it is crucial to make a correct judgment. This process takes a manual workforce and if this task could be automated, resources could be used more efficiently. Automation of many real world processes evolved with the fast development of computer technologies in recent decades. Machine learning and AI-based techniques are present in every aspect of life and they are widely used for automating processes. For signature verification different approaches and techniques where developed over the years and are also used in different systems. Some approaches are briefly described in [3], [4] and [5] and there is much more literature since the problem is still a significant research topic. The problem itself requires methods for image processing in order to get the necessary data on the one hand and on the other hand the problem can be formulated as mathematical classification problem and has to be solved using mathematical methods. Our approach uses a combination of simple and enhanced image processing techniques for feature extraction and a recent method for data preparation. The classification problem is then solved by a classical approach using a **Support Vector Machine** (SVM), which will be described in the following sections.

3 Data Description

The online signatures used in this project are handwritten on a tablet by every member of the team and consist of 25 valid and 25 invalid signatures each. Preparing the signatures for this dataset over a time span of two weeks allowed us to capture as much intra-class variety as possible. Valid samples are shown in figure 1, while invalid ones can be seen in figure 6. It is also important to mention that the subset of invalid signatures not only contains skilled, but also random forgeries.

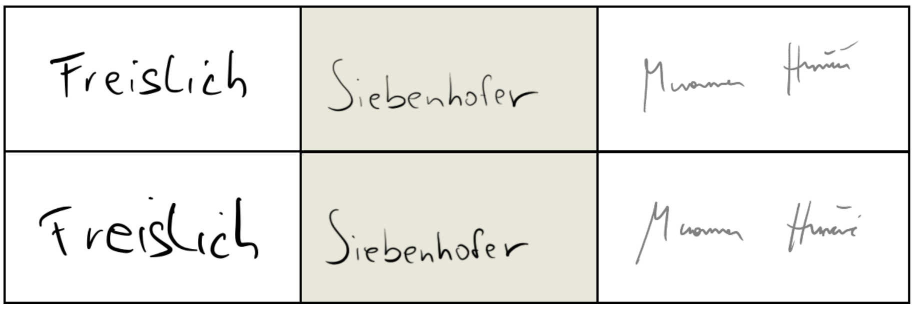


Figure 1: Sample of valid signatures

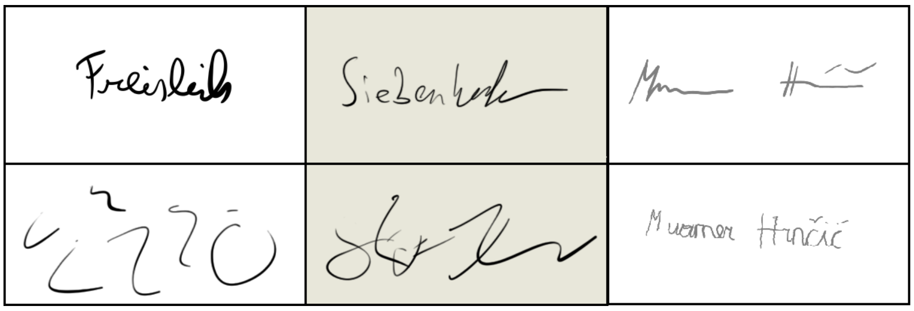


Figure 2: Sample of invalid signatures

4 System Architecture

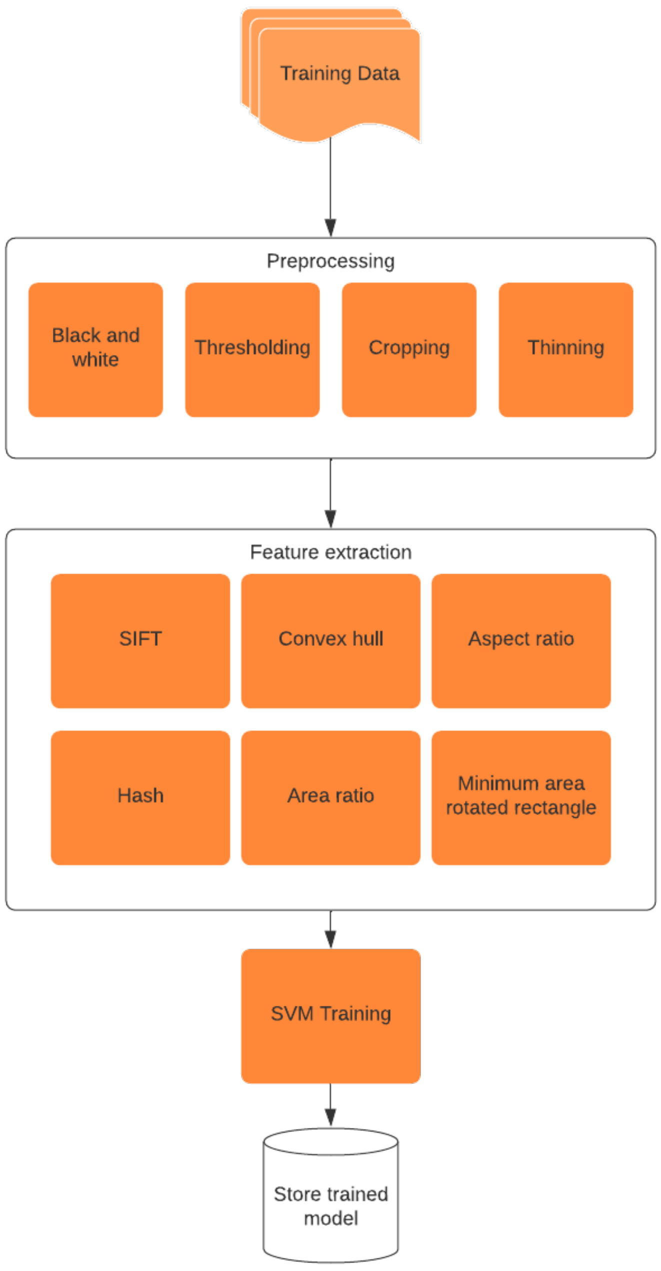


Figure 3: Training Flowchart

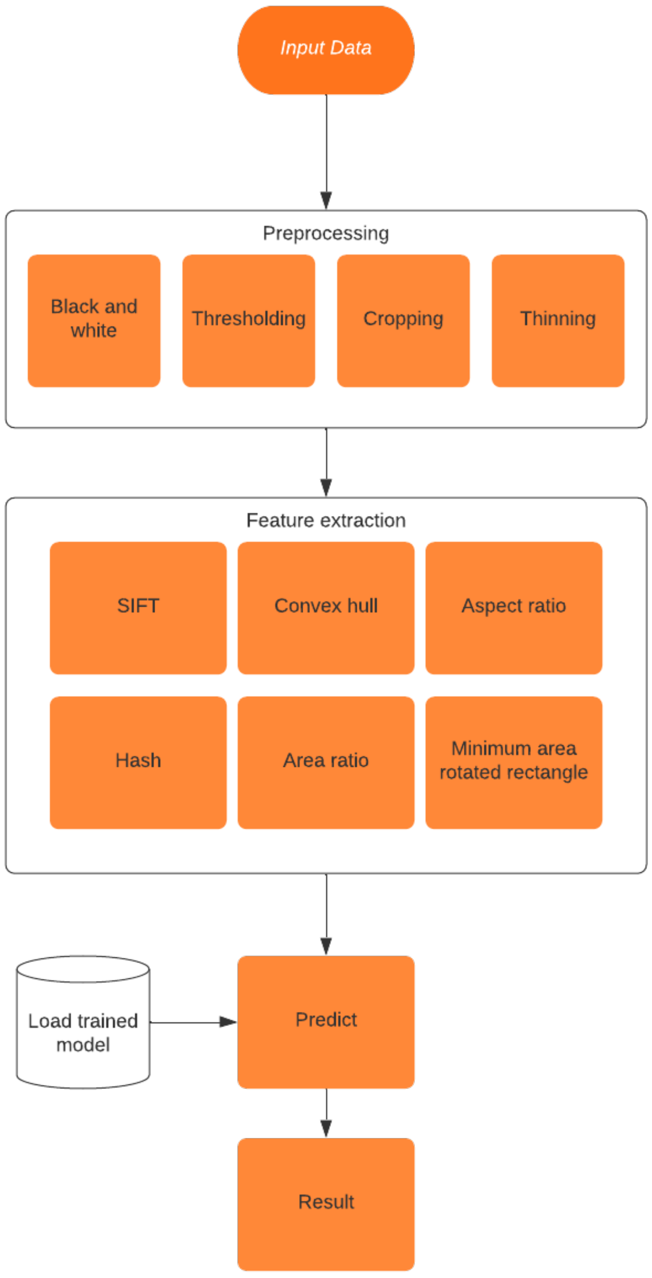


Figure 4: Verification Flowchart

```
Algorithm 1: Train
  Result: Trained Model
  load signatures;
  init featureList;
  for signature in signatures do
      toGrayScale(signature);
      threshold(signature);
      crop(signature);
      thin(signature);
      featureList.add(
        extractSIFTFeatures(signature)
        extractConvexHullFeatures(signature)
        extractAspectRatioFeatures(signature)
        extractHashFeatures(signature)
        extractAreaRatioFeatures(signature)
        extractMinAreaRotatedRectFeatures(signature)
      );
  end
  splitTestAndTrainData();
  testModel();
  saveModel();
```

Figure 5: Training Pseudocode

```
Algorithm 2: Verify Signature
  Result: Signature is valid or invalid
  load signature;
  load trainedModel;
  toGrayScale(signature);
  threshold(signature);
  crop(signature);
  thin(signature);
  signatureFeatures.add(
    extractSIFTFeatures(signature)
    extractConvexHullFeatures(signature)
    extractAspectRatioFeatures(signature)
    extractHashFeatures(signature)
    extractAreaRatioFeatures(signature)
    extractMinAreaRotatedRectFeatures(signature)
  );
  if predict(trainedModel, signature) then
      print("Signature is valid");
  else
      print("Signature is invalid");
  end
```

Figure 6: Verification Pseudocode

5 Implementation

The first step in confirming the authenticity of the given handwritten signature is to preprocess the image in order to increase the efficiency of the verification, to reduce noise and to extract better features used for checking the genuinity of the signature. We start by turning the RGB input image into a black-white image, as seen in figure 7.

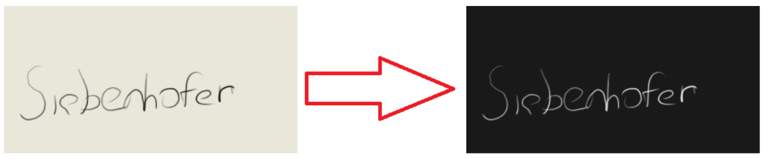


Figure 7: RGB input to black-white

To turn the black and white image into a binary image, we use simple thresholding. After testing multiple values, our conclusion was to go with a threshold value of 30 to produce optimal results.

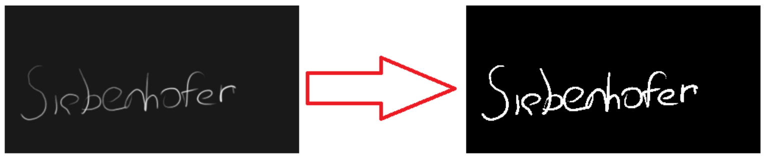


Figure 8: Black-white to binary

To reduce unnecessary information being processed and overall computation time, we decided to crop the image. To solve this task we decided to look for rows and columnccs that have at least one pixel along rows and columns that is greater than 0. Thus, if **img** represents the image data, you would have correspondingly two boolean arrays : (img $>$ 0).any(1) and (img $>$ 0).any(0). We then used those boolean arrays to index into image data with it for the final extracted data, which is the required bounding box data. This process of extracting the region of interest from the binarized signature is shown in figure 9 down below.

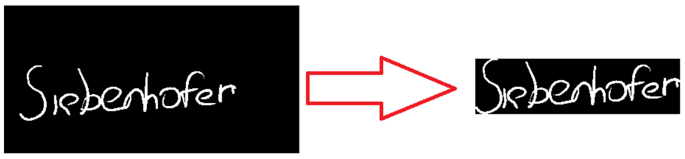


Figure 9: Cropping

To increase the quality of the extracted features, thinning using a cross-shaped kernel with a kernelsize of three was applied to the cropped and binarized signature.

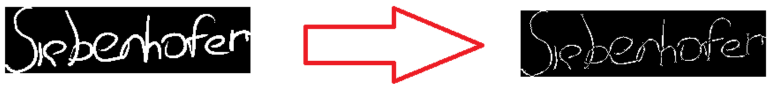


Figure 10: Thinning

After preprocessing the data, we need to extract good features that enable us to determine if the given signature is valid or invalid. We decided to use the aspect ratio, the area ratio between the convex hull of the image and the minimum area rotated rectangle and the area ratio between the contours and the minimum area rotated rectangle, as well as SIFT features to be able to fulfill this task. An example for the minimum area rotated rectangle is shown down below.

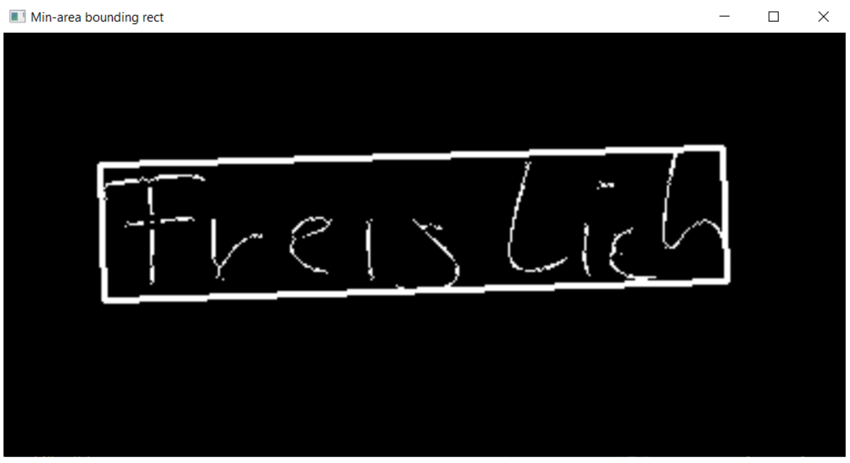


Figure 11: Min-area rotated rectangle

The convex hull, also know as convex envelope or convex closure is the smallest convex set, which contains the shape to be analyzed. Figure 12 shows the convex hull for the same signature used to show the minimal area rotated rectangle in figure 11.

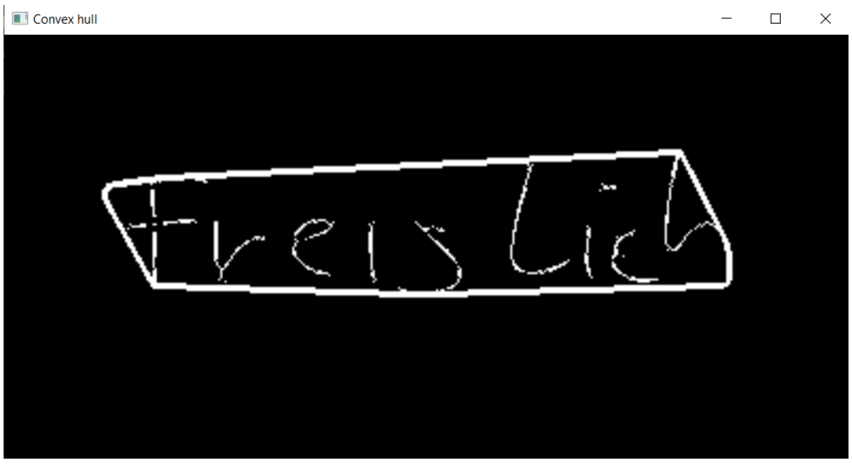


Figure 12: Convex hull

Contours are essentially a curve joining all the continuous points along the boundary, which have the same intensity values. Drawing those contour lines with a thickness of three pixels on top of the signature to analyze results in following output.

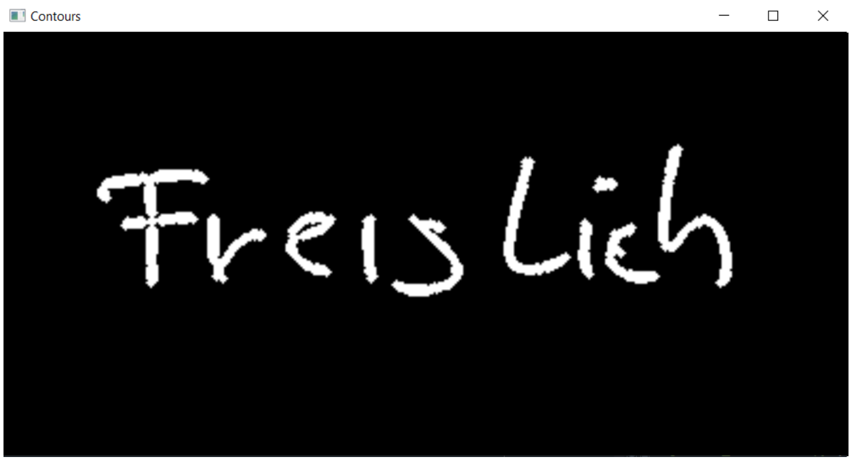


Figure 13: Contours

Another feature we decided to use to check the authenticity of the signatures are the SIFT Keypoints/Descriptors. We decided to use SIFT features because of the invariance to scaling, as well as rotation compared to only rotation invariance for other corner detectors like Harris etc. After locating the keypoiints, a descriptor for each of of them is created. A 16x16 neighbourhood around every keypoint is taken and this space is divided into 16 4x4 sub-blocks. For each of these sub-blocks an orientation histogram with 8 bit resolution is generated. The representation of this histogram as a vector is used to form the keypoint descriptor. Additional measures are taken to achieve robustness against illumination changes, rotation, etc. One example, showing the detected keypoints using SIFT of a signature we are trying to verify, can be seen in figure 14.

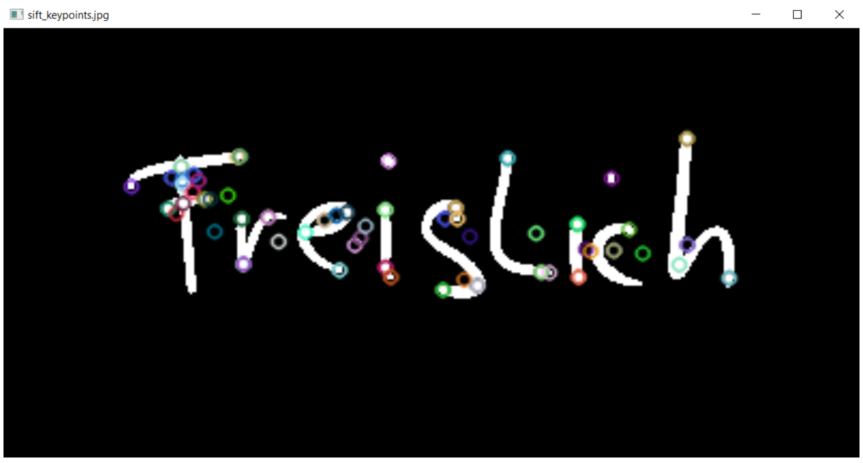


Figure 14: SIFT keypoint detection

All of the aforementioned features are then used to train a linear support vector machine or also called support vector classifier. A support vector machine is a common model to solve classification problems and is extensively described in [6]. The main idea of the model is that we have a data matrix $X = (x_1, x_2, \ldots, x_n)$ and a target vector y. The task is to find a parameter vector w and the intercept b that satisfies $y_i = f(x_i)^T w + b$, where $y_i \in \{-1, 1\}$. The large margin linear classifier with maximum margin is a solution to the problem. This classifier separates the given data in the binary case into two regions where $y_i = 1$ if $w^T x_i + b > 0$ and $y_i = -1$ if $w^T x_i + b < 0$. After a scale transformation these conditions are equivalent to $y_i = 1$ if $w^T x_i + b \geq 1$ and $y_i = -1$ if $w^T x_i + b \leq -1$. A visualization of the problem can be seen in 15.

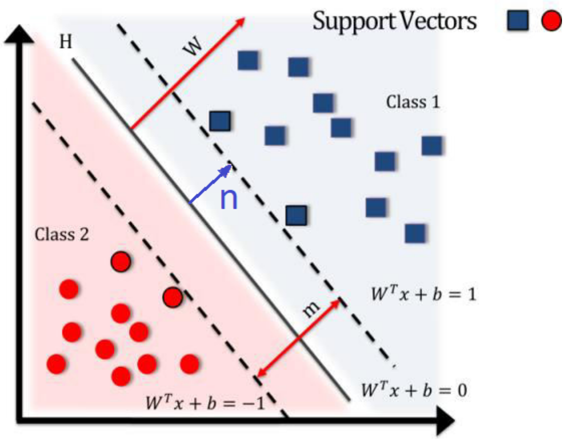


Figure 15: Large Margin Linear Classifier

For support vectors x^+ and x^- we know that $w^T x^+ + b = 1$ and $w^T x^- + b = -1$ and the margin width can be found as $m = (x^+ - x^-) \cdot n = (x^+ - x^-) \cdot \frac{w}{\|w\|} = \frac{2}{\|w\|}$, where n is the normal vector on the line $w^T x + b = 0$. We come to the maximization problem

Model:
$\max \frac{2}{\|w\|}$
s.t.

(1) $y_i = 1$ if $w^T x_i + b \geq 1$

(2) $y_i = -1$ if $w^T x_i + b \leq -1$

or the minimization variant

Model:
$\min \frac{\|w\|^2}{2}$
s.t.

(1) $y_i = 1$ if $w^T x_i + b \geq 1$

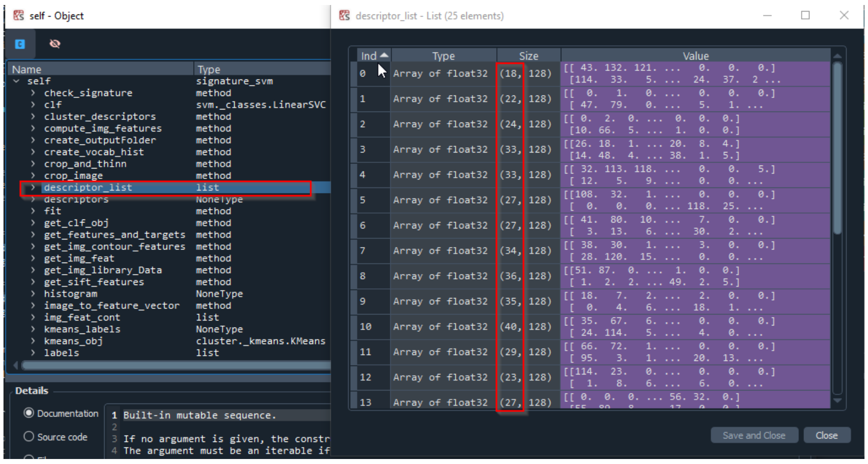
(2) $y_i = -1$ if $w^T x_i + b \leq -1$

which can be solved in different ways as described in the literature. We use the **LinearSVC** implementation of the library **sklearn.svm** in the following simple way.

```
from sklearn.svm import LinearSVC
...
#Linear SVC:
#Create Linear support vector classifier with vocabulary as
features and
#two classes target, (1,2) = ("valid","invalid")
def fit(self, X_train, y_train):
    #Copy train features
    self.X_train = X_train.copy()
    #Copy train target
    self.y_train = y_train.copy()
    #Fit the model
    self.clf.fit(self.X_train, self.y_train)
    #Mark the model as trained
    self.trained = True

def predict(self, X):
    #Predict class for feature X
    return self.clf.predict(X)

def score(self, X_test, y_test):
    #Return mean accuracy of the test data set
    return self.clf.score(X_test, y_test)

def get_clf_obj(self):
    #Return the classifier object
    return self.clf
...
```

Listing 1: Linear SVC

What we can see here is that only the training data X, and the target vector y are required to fit the model. But before doing so we first have to construct the training data. We know in advance which signatures are valid and invalid so the construction of the target vector is no problem at all. We want to use the above mentioned features as data matrix. One way would be to create X as matrix of all found features. The problem with this is shown in 16. It shows the list of SIFT descriptors for valid images of one user.



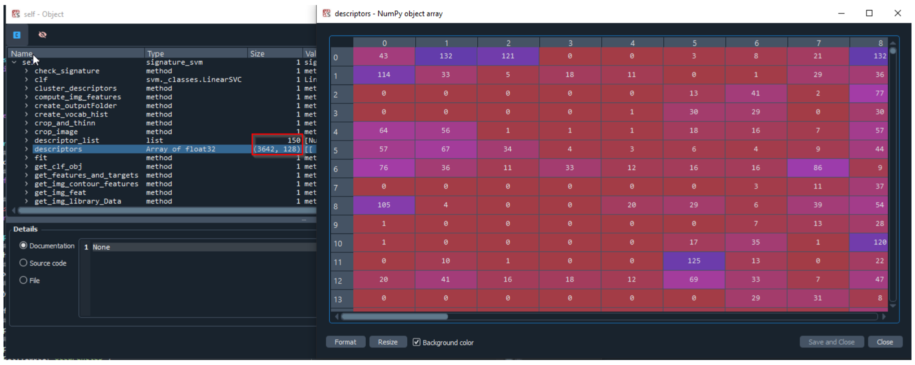
Name	Type
self	signature_svm
check_signature	method
clf	svm._classes.LinearSVC
cluster_descriptors	method
compute_img_features	method
create_outputFolder	method
create_vocab_hist	method
crop_and_thinn	method
crop_image	method
descriptor_list	list
descriptors	NoneType
fit	method
get_clf_obj	method
get_features_and_targets	method
get_img_contour_features	method
get_img_feat	method
get_img_library_Data	method
get_sift_features	method
histogram	NoneType
image_to_feature_vector	method
img_feat_cont	list
kmeans_labels	NoneType
kmeans_obj	cluster._kmeans.KMeans
labels	list

```
1 Built-in mutable sequence.
2
3 If no argument is given, the constr
4 The argument must be an iterable if
```

Ind	Type	Size	Value
0	Array of float32	(18, 128)	[[43. 132. 121. ... 0. 0. 0.] [114. 33. 5. ... 24. 37. 2 ...
1	Array of float32	(22, 128)	[[0. 1. 0. ... 0. 0. 0.] [47. 79. 0. ... 5. 1. ...
2	Array of float32	(24, 128)	[[0. 2. 0. ... 0. 0. 0.] [10. 66. 5. ... 1. 0. 0.]
3	Array of float32	(33, 128)	[[26. 18. 1. ... 20. 8. 4.] [14. 48. 4. ... 38. 1. 5.]
4	Array of float32	(33, 128)	[[32. 113. 118. ... 0. 0. 5.] [12. 5. 9. ... 0. 0. ...
5	Array of float32	(27, 128)	[[108. 32. 1. ... 0. 0. 0.] [0. 0. 0. ... 118. 25. ...
6	Array of float32	(27, 128)	[[41. 80. 10. ... 7. 0. 0.] [3. 13. 6. ... 30. 2. ...
7	Array of float32	(34, 128)	[[38. 30. 1. ... 3. 0. 0.] [28. 120. 15. ... 0. 0. ...
8	Array of float32	(36, 128)	[[51. 87. 0. ... 1. 0. 0.] [1. 2. 2. ... 49. 2. 5.]
9	Array of float32	(35, 128)	[[18. 7. 2. ... 2. 0. 0.] [0. 4. 6. ... 18. 1. ...
10	Array of float32	(40, 128)	[[35. 67. 6. ... 0. 0. 0.] [24. 114. 5. ... 4. 0. ...
11	Array of float32	(29, 128)	[[66. 72. 1. ... 0. 0. 0.] [95. 3. 1. ... 20. 13. ...
12	Array of float32	(23, 128)	[[114. 23. 0. ... 0. 0. 0.] [1. 8. 6. ... 6. 0. ...
13	Array of float32	(27, 128)	[[0. 0. 0. ... 56. 32. 0.]

Figure 16: Descriptor list for one user

Every descriptor has 128 columns which is totally fine but one fact is that not all descriptors have the same number of rows. Another problem is that if we get all features of all images the data matrix gets large even for small number of users which is shown in 17. At the time of implementation only three users with a total of 150 pictures were in the present image library for model training. Using this approach the size of the data matrix would increase rapidly through adding new users and their valid-invalid signature images.



Name	Type	Size	Val
self	signature_svm	1	sig
check_signature	method	1	met
clf	svm._classes.LinearSVC	1	Lin
cluster_descriptors	method	1	met
compute_img_features	method	1	met
create_outputFolder	method	1	met
create_vocab_hist	method	1	met
crop_and_thinn	method	1	met
crop_image	method	1	met
descriptor_list	list	150	[Nu
descriptors	Array of float32	(3642, 128)	[[
fit	method	1	met
get_clf_obj	method	1	met
get_features_and_targets	method	1	met
get_img_contour_features	method	1	met
get_img_feat	method	1	met
get_img_library_Data	method	1	met

```
1 None
```

	0	1	2	3	4	5	6	7	8
0	43	132	121	0	0	3	8	21	132
1	114	33	5	18	11	0	1	29	36
2	0	0	0	0	0	13	41	2	77
3	0	0	0	0	1	30	29	0	30
4	64	56	1	1	1	18	16	7	57
5	57	67	34	4	3	6	4	9	44
6	76	36	11	33	12	16	16	86	9
7	0	0	0	0	0	0	3	11	37
8	105	4	0	0	20	29	6	39	54
9	1	0	0	0	0	0	7	13	28
10	1	0	0	0	0	17	35	1	120
11	0	10	1	0	0	125	13	0	22
12	20	41	16	18	12	69	33	7	47
13	0	0	0	0	0	0	29	31	8

Figure 17: Descriptor list for all users

To overcome this problem we use a technique called **bag of visual words** (BoVW) described in [7] to reduce data in an intelligent and convenient way. BoWV is a common method for image classification. The main idea reflects the idea of the bag of words techniqe in natural language processing. BoVW treats features as words. First we extract and gather all the features as described above. After this the feature space is quantized by clustering techniques. We use the **kmeans** method, described in [6], to cluster the feature space where the resulting centroids form the visual dictionary vocabulary. With the vocabularies for each image equipped a frequency histogram is created and we get a big part of the data X which is now much more compact, without loosing to much information. The process is described in 18 and the resulting histogram when using 120 clusters is shown in 19.

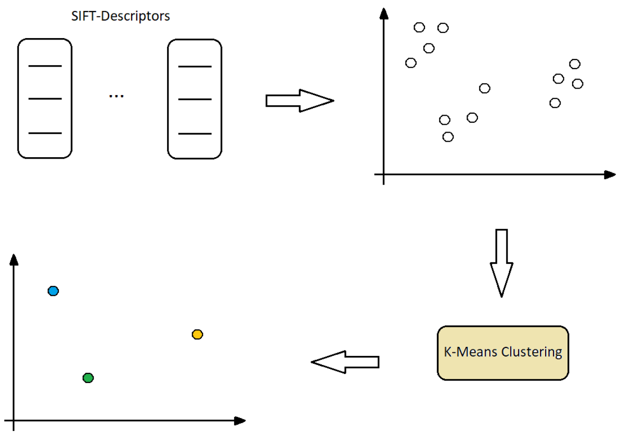


Figure 18: BoVW for SIFT Descriptors

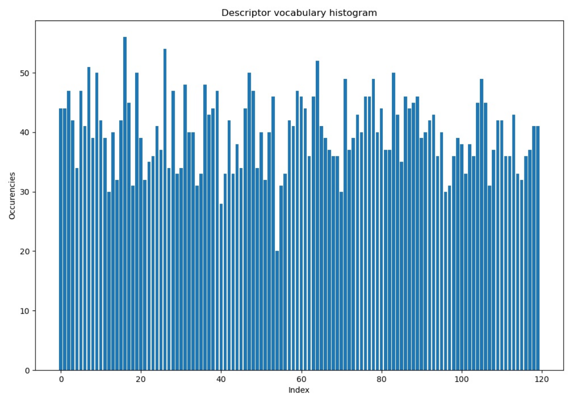


Figure 19: Descriptor vocabulary Histogram

After getting the histogram we finally combine it with the four additional features as columns, normalize the data and obtain the data matrix X. Now the model can be fitted and new signatures can be classified with a certain accuracy. After fitting the model we save the model for later use. A demonstration of how the model classifies is implemented in **main.py** and the results are shown in figures 20 and 21 .

```python
import cv2 as cv
from matplotlib import pyplot as plt
from signature_svm import load_svm_from_datafile

if __name__ == "__main__":
    #Path to image
    image_path = "testing/val_f.png"

    #Load test image
    image = cv.imread(image_path)
    #Load trained signature_svm
    sigSVM= load_svm_from_datafile()

    #Transform image to feature vector
    #(vocabulary + special features)
    X_img = sigSVM.image_to_feature_vector(image_path)
    y_hat = sigSVM.predict(X_img)

    print("Predicted class for image:")
    print(int(y_hat))
    result = sigSVM.check_signature(image_path)

    print("Signature is " + result)
```
Listing 2: Testing the model

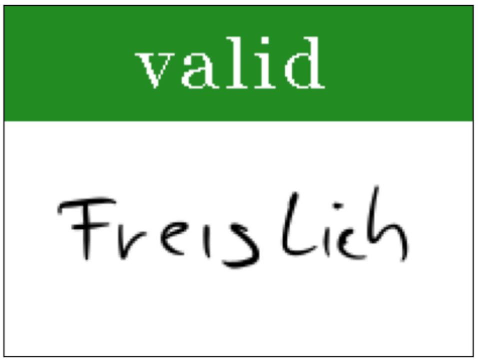
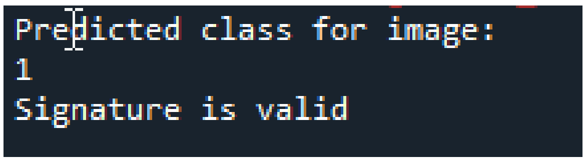


Figure 20: Classified signature



```
Predicted class for image:
1
Signature is valid
```

Figure 21: Output of method predict()

The next part was to test the implementation and to find good parameters.

6 Performance and model evaluation

Training and testing the model was done in different ways to get a good feeling of how the model behaves and how accurate it is with classifying present images and new images. First of all we performed simple training where we used 80% of the image set for training and the remaining for testing. We obtain the confusion matrix in 22 and the performance metrics in 23.

```python
if __name__ == "__main__":
    ##########################################
    ...

    #Create signature SVM with 120 clusters for feature
    #clustering and a given random seed
    sigSVM = signature_svm(n_clusters=120, random_state =
    randomCharSeed, write_output=True)

    #Get the feature matrix and the target (labels) of
    #all images in the signature folder
    X, y = sigSVM.get_img_library_Data()

    #Split the data into train and test data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, train_size=size, random_state=randomCharSeed)

    #Fit the model
    sigSVM.fit(X_train, y_train)

    ...
```

Listing 3: Training simple

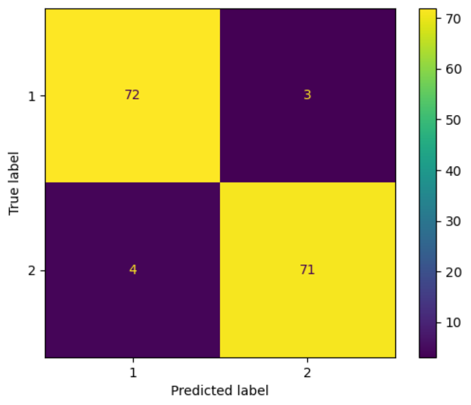
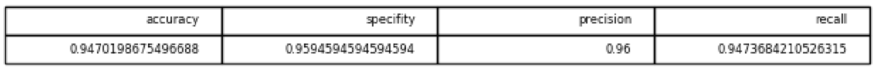


Figure 22: Confusion matrix



accuracy	specifity	precision	recall
0.9470198675496688	0.9594594594594594	0.96	0.9473684210526315

Figure 23: Performance metrics

Since the overall performance should not depend on the choice of feature vectors for training we also performed a K-fold cross validation to get the mean performance of the model. K-fold cross validation is an iterative method for testing models where a certain percentage of the overall data is used as training data and the remaining data is used as hold out and/or test data. In every iteration the data is shuffled and split into train and test data and the model is fitted. This is performed several times and the overall performance of the model is the mean performance of all iterations. This is implemented as follows and the results are shown in 24.

```
if __name__ == "__main__":
    #########################################
    ...
    #Exhaustive Testing, Cross validation, for train size 80%
    #and 120 feature clusters
    iterations = 20S
    #List to store the scores
    scores = []
    #Copy of features and target
    X_cross = X.copy()
    y_cross = y.copy()

    for i in range(iterations):

        #Shuffle the data
        X_cross, y_cross = shuffle(X_cross,y_cross,
        random_state=randomCharSeed)
        #Create signature SVM, no output is need at this point
        sigSVM = signature_svm(n_clusters=120, write_output=False)

        #Split the data into train and test data
        X_train, X_test, y_train, y_test =
            train_test_split(X_cross, y_cross,
            train_size=size, random_state=randomCharSeed)

        #Fit the model
        sigSVM.fit(X_train, y_train)

        #Append score
        scores.append(sigSVM.score(X_test, y_test))

    #Save mean score
    mean_score = np.mean(scores)
    ...
```

Listing 4: K-fold cross validation

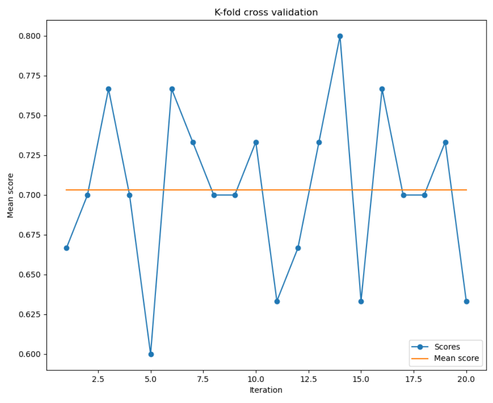


Figure 24: Cross validation, 20 iterations

The next important topic is the choice of parameters. Relevant model parameters are the number of clusters used for BoVW and the train size. Other parameters were already fixed before and did not change during testing. A loop was created to iterate over different values of train set sizes and cluster sizes. This test is implemented as follows.

```
if __name__ == "__main__":
    ########################################
    ...

    #Exhaustive Testing, Train size and cluster size
    #are variable
    train_size = np.arange(0.1, 1, 0.1)
    cluster_size = np.arange(20, 130, 20)
    #Copy data for training and testing
    X_ex = X.copy()
    y_ex = y.copy()

    #Overall scores
    scores_ex = []
    for csize in cluster_size:
        #Scores for actual size
        score_act_size = []
        for tsize in train_size:
            #Shuffle the data
            X_ex, y_ex = shuffle(X_ex,y_ex,
            random_state=randomCharSeed)
            #Create signature SVM, no output is need at this point
            sigSVM = signature_svm(n_clusters=csize,
            write_output=False)

            #Split the data into train and test data
            X_train, X_test, y_train, y_test =
                train_test_split(X_ex, y_ex,
                train_size=tsize, random_state=randomCharSeed)

            #Fit the model
            sigSVM.fit(X_train, y_train)

            #Append score
            score_act_size.append(sigSVM.score(X_test, y_test))
        #Append scores for actual cluster size
        scores_ex.append(score_act_size)
    ...
```

Listing 5: Parameter search

The image 25 shows the results of this test.

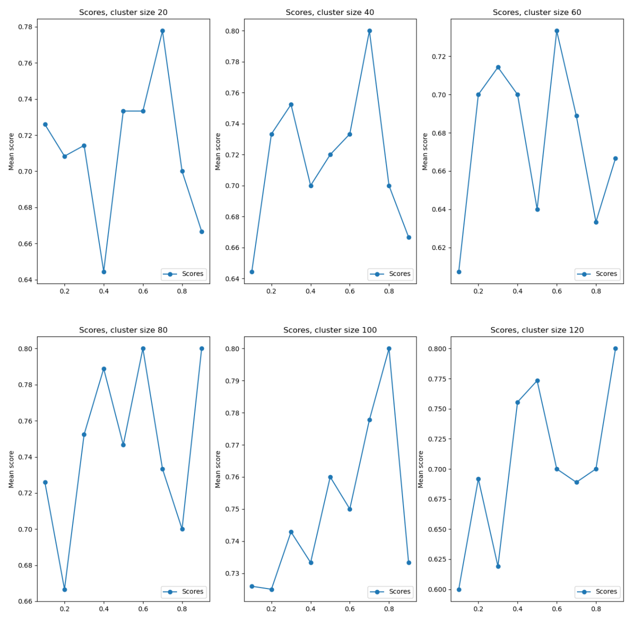


Figure 25: Parameter search

The first things one can verify is that small train sizes on average do not bring good performance which is clear since the model must have enough information of what is a valid and what is an invalid signature. It is also clear that accuracy increases with increasing train size. One interesting observation is the impact of the cluster size. A bigger cluster size increases the average performance and the best accuracy is obtained for test sizes between 60% and 80%. So our choice for the default parameters of train size 80% and cluster size 120 are reasonable.

7 Discussion

Our model behaves sufficiently good with the created signatures and the overall accuracy is high. One way to increase the accuracy is to get more images per user which again leads to the problem of bigger feature matrices and longer training times. We choose to thin the signatures per default. Nevertheless sometimes thinning processes remove good features which could lead to better accuracy. Therefore we tested the model also without thinning. Feature extraction takes longer since more keypoints can be detected and therefore more descriptors can be computed. Nevertheless if we use again 120 clusters for BoVW the histogramm dimensions remain the same but performance is increased which can be seen in 26 and 27. The k-fold cross validation in 28 also shows that the average score is now higher than before. Depending on the amount of images per user and the number of users it should be clear which variant should be taken. If the number of users and/or images per user increases maybe it would be a good workflow to take a combination of thinned and not thinned images.

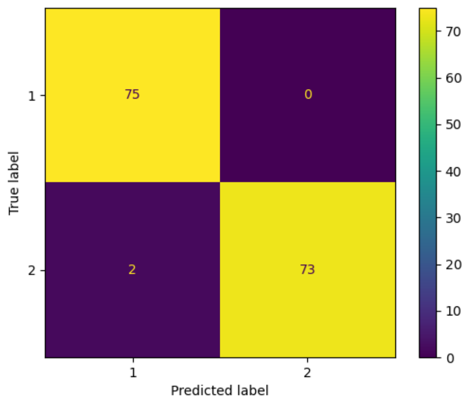
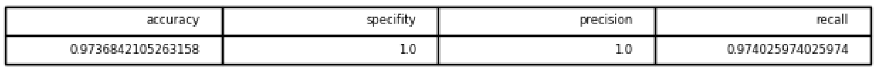


Figure 26: Confusion matrix, no thinning



accuracy	specifity	precision	recall
0.9736842105263158	1.0	1.0	0.974025974025974

Figure 27: Performance metrics, no thinning

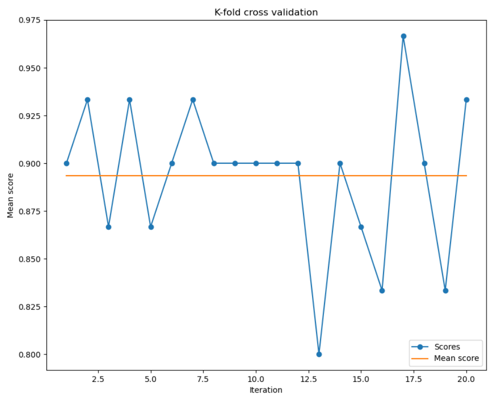


Figure 28: K-fold cross validation, 20 iterations, no thinning

8 Conclusion

Signature verification is an interesting problem which can be tackled by different approaches and most approaches use neuronal networks, convolutional networks and other models to solve the problem. Our implementation uses a handful of selected image processing techniques and data preparation methods to train a classical linear SVM and the results are really good for most requirements.

References

[1] Réjean Plamondon and Sargur N Srihari. Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on pattern analysis and machine intelligence*, 22(1):63–84, 2000.

[2] Luiz G Hafemann, Robert Sabourin, and Luiz S Oliveira. Analyzing features learned for offline signature verification using deep cnns. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2989–2994. IEEE, 2016.

[3] Donato Impedovo and Giuseppe Pirlo. Automatic signature verification: The state of the art. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38:609 – 635, 10 2008.

[4] Luiz Gustavo Hafemann, Robert Sabourin, and Luiz Soares de Oliveira. Offline handwritten signature verification - literature review. 11 2017.

[5] Saba Mushtaq and Ajaz Mir. Signature verification: A study. pages 258–263, 09 2013.

[6] Christopher M. Bishop. Pattern recognition and machine learning (information science and statistics). 2007.

[7] Gabriela Csurka, Christopher Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. Visual categorization with bags of keypoints. *Work Stat Learn Comput Vision, ECCV*, Vol. 1, 01 2004.