Machine learning



**Group 40**

Name: Stefan Winter

Student number: 2067606

Name: Joost Oudesluijs

Student number: 2056392

Name: Joost Schutte

Student number: 2069625

[880083-M-6](https://tilburguniversity.instructure.com/courses/5509)

Dr. Güven Ç and Dr. Önal, I.

Department of Cognitive science and Artificial intelligence

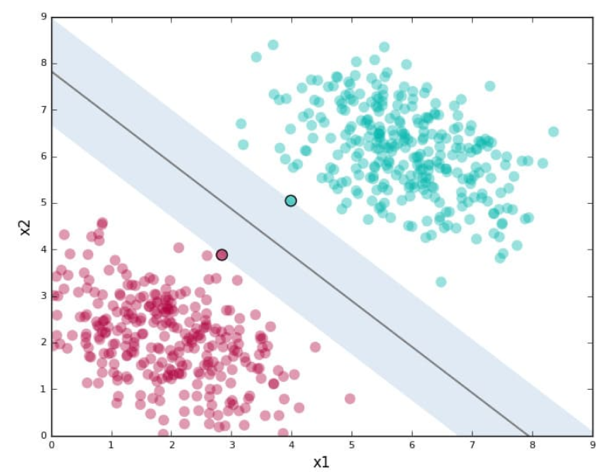
School of Humanities and Digital sciences

Tilburg University

June 2021

|  |
| --- |
|  |

# Task 1

**Model 1**

**Support vector machine**

The Support Vector Machine (SVM) is an algorithm that fits a hyperplane to classify all datapoints to categories in an N-dimensional space (Ghandi, 2018). This hyperplane (the black line in the image below) maximizes the distance or margin (light blue area) between the different classes. The Support Vectors are the closest points to the classifier and determine the position of the hyperplane.

Figure : Support Vector Machine visualized

To correctly classify the datapoints, a kernel function projects low-dimensional data into a higher-dimensional space. Enhancing the dimensions makes it possible to appropriately divide the observations into different classes. Therefore, Support Vector Machines work more effectively in higher dimensional spaces, or in datasets with a high number of input features (Ghandi, 2018). Since SVM can handle multiclass classification problems and works well with a high number of input features, this model is applied for task 1.

**Input to classifier**

To represent the image data from training data and test data, the vectors are standardized with the StandardScaler function from scikit-learn. What this function does is subtract the mean from every observation and divide that by its standard deviation. Particularly for SVM, the algorithm assumes that all features are centered around zero and have variance in the same order (scikit-learn, 2021). The data does not need to be split into a training and a validation set because using cross-validation and grid search for parameter tuning handles this automatically.

**Hyperparameter tuning**

To optimize this model, a pipeline is created in which multiple hyperparameter settings are evaluated. This is done using the GridSearchCV function from scikit-learn, where five-fold cross-validation is applied. This means that every part of the training data is used four times for training and one time for validation purposes (Souza, Matwin, Japkowicz, 2002). In the first part of the pipeline, Principal Component Analysis (PCA) is applied to reduce the dimensionality and the noise of the data (Husson et al., 2010). During the grid search, the n\_components parameter for PCA ranged from 75 to 100 percent with increments of 1 percent. The result of this is that 90 percent of all variance can be explained using only 58 features, which is a reduction of 726 dimensions. The cost parameter (C) is the second parameter that was tuned to find a good number of samples that is allowed in the margin, to find the lowest overall error (scikit-learn, 2021). Values between 1 and 6 are used, with increments of 1. To transform low dimensional input to a higher dimensional space, multiple kernels can be applied; a polynomial function, the Radial Basis Function (RBF) and a sigmoid function.

**Model training**

The obtained training score after hyperparameter tuning is around 0.99. The best settings for PCA turned out to be a dimensionality of 91 percent variance explained. This reduces the 784 features to only 65 features. The best value for parameter C is 5. Lastly, the best SVM kernel is RBF. This makes sense since the RBF kernel and the StandardScaler both squeeze their observations to a (near) Gaussian distribution. Therefore, the RBF kernel should be the best fit (scikit-learn, 2021).

**Results**

The hyperplane that resulted from training the model, with the appropriate hyperparameter settings, gives an accuracy of 0.88 on the test set. This means that almost 90 percent of all test images are classified correctly (see Appendix 2 for the classification report). The most difficult label to classify is the letter R. The letters V, Y, K, U and W were most often incorrectly classified as an R with 38, 35, 29, 27 and 21 times respectively. R itself was classified incorrectly as U and V most often; 35 and 21 times respectively. Thus, the main problem with R seems to be with false positives, which is a precision problem.

**Model 2**

**Random forest**

Random Forest (RF) is based on decision trees. Decision trees are a supervised learning model and are often used for classification tasks (James et al., 2013). The architecture of the decision tree is comparable to a regular tree; it has roots, nodes, branches, and leaves. The root node is the starting point of the tree and splits every node with recursive partitioning based on the largest information gain (Sato & Tsukimoto, 2001). At each coming leaf node, the data is split into partitions based on the feature with the second largest information gain. This process is repeated until a base case is reached or when the label is small enough given the min\_samples\_leaf parameter input.

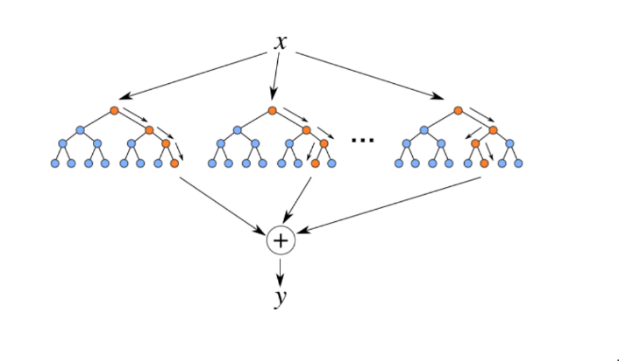
Although decision trees are transparent and easily interpretable, this model is quite prone to overfitting. To overcome this problem, Random Forests are introduced. RFs are different from decision trees in the sense that they generate multiple trees. The process of generating multiple trees is done by randomly sampling the training data and random subsets of features when splitting nodes (Koehrsen, 2018). The random selection of features in each node allows the features that would otherwise be overlooked, to be used in the model (Strobl et al., 2008). The model is then trained with these random samples and subsets to decrease the total variance. However, different combinations of features could generate different predictions, which means that the final prediction is based on a majority vote.

Figure 2: Random Forest Classifier visualized

**Hyperparameter tuning**

The input to the classifier is the same as for the SVM model, so it will not be discussed here again. To increase performance of the RF, hyperparameters can be tuned in multiple ways. To find the optimal number of decision trees that comprise the random forest, the n\_estimator parameter is included in the grid search. To prevent a situation in which every leave is pure and to prevent potential overfitting, the max\_depth parameter is tuned. To consider different splits in features, the max\_features parameter is also tuned. To set a penalty at the minimum number of samples used per leaf node, the min\_samples\_leaf parameter is included. The parameter min\_samples\_split is also included to set a baseline for the minimum number of samples required to split an internal node. The grid search to find the optimal hyperparameters is added to a pipeline. GridSearchCV is again used, with a range from 1 to 20 with increments of 1, and a cross-validation of 5. The n\_components parameter for PCA is also included in the grid search, with a range from 80 to 100 percent explained variance with increment of 1. Lastly, an untuned model without PCA was also executed.

**Model training**

The best settings for the RandomForestClassifier from scikit-learn was the untuned model, with a training set accuracy of approximately 100 percent. The second-best model is the tuned version, the hyperparameter settings of which can be found in Appendix 3.

**Results**

The untuned model gives an accuracy of approximately 0.81 on the test set, which is 0.11 higher than the best version of the tuned model. Again, the most difficult label to classify is the letter R. The same letters are most often incorrectly classified for R as was the case for the SVM model. However, the order in which they occur, based on frequency, differs a bit. Furthermore, the letters S and W are also difficult to classify, but still perform slightly better than R (see Appendix 4). What these three letters all have in common is that they show higher false positives rates, which leads to worse precision when compared to recall.

**Comparison between models**

The SVM model shows a better test performance than the RF model. The accuracy scores between the two models show a difference of 0.08 accuracy in total. This difference is mainly caused by the letters S and W that perform approximately 20 percentage points worse on the RF model. Other letters performed roughly the same; within 5 percentage points of each other. A remarkable finding is that both models seem to have problems to correctly classify the letter R. Furthermore, both models tend to show higher rates of false positives, and therefore have a lower precision, on the letters with low test set scores. The last comparison between the models is runtime. The time it takes to train the Random Forest model is only 15 seconds, whereas the SVM model takes four times longer. However, this is only a small absolute difference, which means that the SVM model is the overall best option.

# Task 2

The model we used for task 2 is the Support Vector Machine model. This model uses 61 features with an explained variance of 0.91, uses the RBF kernel and has an accuracy score of 0.88 on the test set from task 1. To accurately predict the labels from the images in the test set of task 2, several pre-processing steps need to be performed.

The first step in the pre-processing stage is to apply the function “hand\_finder”. This function is created to extract the index of the top-left pixel of each hand that can be found in the images. To do so, Canny edge detection is applied to the image, which transforms the image into something in which the hands are more clearly visible in the image. The output of the edge detection algorithm is used to look for when a hand appears in an image. This seemed to be the case when at least 10 values in the first row of the output had a value of 0. The index of the first value was then extracted and saved to a list corresponding to that image. As a result of the hand\_finder function, we were left with a list of lists, where each list represented an image, and the list contained the index of the top-left pixel of each hand in that image.

The second part consists of the “hand\_locator” function that we created. The goal of this function is to extract the 28x28 hand images, starting at the top-left pixel for every hand in the full image. These top-left pixel indices were obtained using the hand\_finder function. To get the best possible result, we decided that we wanted to extract 5 different 28x28 images for each hand in the full image. This is because the hand\_finder function was not 100 percent accurate. The way this is done is by taking the index of the top-left pixel of the hand, and adding a value ranging between -3 and 2 to the index. These values were chosen because it seemed that the hand\_finder function overshot the top-left pixel consistently, so 1 was always deducted from this. Using the -3 to 2 range means that the middle value is -1, which lines up with what we wanted. The rest of the function took the index value from the hand\_finder function, added the value from the range to it, and then used that value to extract the 27 pixels to its right and the 27 pixels going down. This resulted in the final 28x28 hand images, the values of which were added to a list. What the hand\_locator function returns, is again a list of lists, where each list again represented an image, and the list contained all values of the hands that were found in that image.

The last function we created is the “predict” function. The goal of this function is to generate five predictions for each hand. This function simply iterates over the output of the hand\_locator function and creates a prediction for each hand (list of values) inside the image (the larger list of lists). This prediction is then added to a new list, which in the end will have five predictions for each hand. The rest of the function is simply to add the predictions for each hand together to get predictions for the full image instead of only the hand. The output of the predict function is a list of lists, where each list represents an image, and the list contains 5 predictions for that image.

Now that we have the predictions for all images, it is time to check the accuracy of this method. To do so, we imported the true labels of the sample dataset using a function that reads the file name of each image. These labels were then used to check the accuracy of our predictions for the images in the sample dataset.

The accuracy of the Support Vector Machine model, with the parameter settings from task 1, generated a score of 0.60 on the sample dataset. This means that 14 images were correctly classified, while 11 images were not. Although this accuracy is still not as high as the accuracy that we achieved in task 1, our model outperforms random guessing. Random guessing would result in a probability of 0.042 (1/24) for each hand, to the power of the number of hands in the image. So if we had two hands in one image the probability of random guessing would be 0.00178 (0.042^2). Hence our model predicts a lot better than random guessing, but could still be enhanced by 28 percent points. This is the difference between the result from the SVM model (0.88) from task 1 and the performance of the model (.60) on task 2 data. Therefore the model is not perfect, but does a decent job overall.

Finally, the predictions for all 10,000 images from the test set were made and saved into a csv file. This process of predicting the images took approximately 5 minutes. To make a final comparison, the Random Forest model, with the appropriate parameter settings, generated an accuracy of 60 percent for task 2 as well. However, the running time took 6 times longer than the SVM model. Therefore, looking at efficiency, the SVM model outperforms the RF model once again.

# Reference

Breiman, L. (2001). Random forests. Machine learning, 45, 5-32.

Gandhi, R. (2018b, July 5). Support Vector Machine — Introduction to Machine Learning Algorithms. Retrieved from https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

Koehrsen, W. (2018, December 1). An Implementation and Explanation of the Random Forest in Python. Retrieved from <https://towardsdatascience.com/an-implementation-and-> explanation-of-the-random-forest-in-python-77bf308a9b76

Sato, M. and H. Tsukimoto. 2001. Rule extraction from neural networks via decision tree induction. In Proceedings of the International Joint Conference on Neural Networks.

Strobl, C., Boulesteix, A. L., Kneib, T., Augustin, T., & Zeileis, A. (2008). Conditional variable importance for random forests. BMC bioinformatics, 9, 307.

Scikit-learn (2021). [sklearn.preprocessing](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing).StandardScaler. Retrieved from <https://scikit-learn.org/stable/modules/preprocessing.html>

Souza, J., Matwin, S., Japkowicz, N. (2002). *Evaluating Data Mining Models: A Pattern Language* School of Information Technology and Engineering University of Ottawa.

# Appendices

## 1 Group work

Stefan Winter

* Explaining SourceTree for working together
* Generating the Support Vector Machine model and applying GridSearch for task 1
* Feedback on the report for task 1 and task 2

Joost Oudesluijs

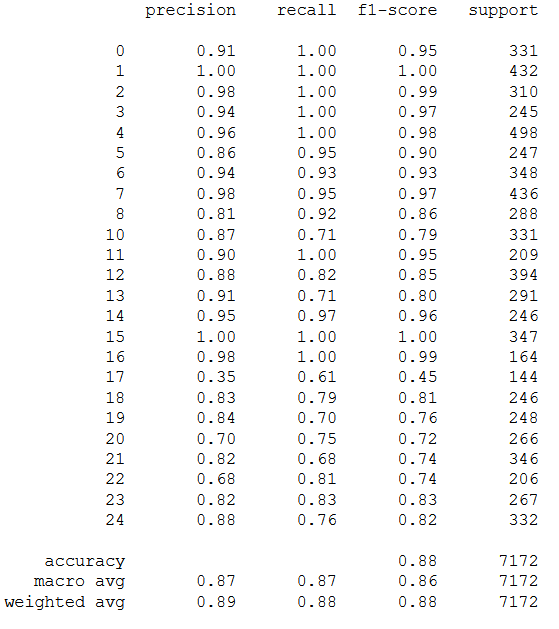
* Generating the Neural Network for task 1
* Writing the report for task 1 and task 2
* Adding the appendices and references for the report

Joost Schutte

* Generating the Random Forest model and applying GridSearch for task 1
* Generating all code (functions and the model) for task 2
* Feedback on the report for task 1 and task 2

## 2 Classification report Support Vector Machine

Table 1: Classification report with the best hyperparameter settings for the SVM model



## 3 Best parameter values for the tuned Random Forest model

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Tuned parameter scores** | **Test set score** |
| n\_components (PCA)  max\_depth  max\_features  min\_samples\_leaf  min\_samples\_split  n\_estimators | 0.85  12  5  1  2  19 | 0.70 |

## 4 Classification report for the Random Forest model

