Machine learning lab assignment 4 Perceptron classifier and cross-validation

Kevin Serrano, *EMARO+, UNIGE*

***Abstract*—In machine learning, the perceptron is an algorithm for supervised learning of binary classifiers. This report will introduce the traditional Rosenblatt perceptron and point out its strengths and weaknesses based on graphical interpretations and performance metrics obtained from k-fold cross-validation. A comparison with the previously seen k-NN algorithm is also carried out.**

***Index Terms*—perceptron, classifier, machine learning, cross-validation.**

I. Introduction

A. Perceptron classifier

The perceptron forms part of the **artificial neural** network models, taking inspiration from the brain.

The perceptron is a basic processing element. It has inputs that may come from the environment (available data) or be the outputs of other perceptrons. Associated with each input, there is a *connection weight*, and the output, in the simplest case, is a weighted sum of these inputs. However, in the most common case, this weighted sum is the input to an *activation function* $f(x)$.

An intercept value w_0 is added to make the model more general and it can ba also thought of as a threshold value for the activation function. It is generally modeled as the weight coming from an extra *bias unit* x_0 which is always $+1$. Thus, obtaining a simplified expression for the perceptron output y.

$$y = f(\boldsymbol{w}^T\boldsymbol{x}) \tag{1}$$

where

$$\begin{cases} \boldsymbol{w} &= \begin{bmatrix} w_0 & w_1 & w_2 & \dots & w_d \end{bmatrix} \\ \boldsymbol{x} &= \begin{bmatrix} 1 & x_1 & x_2 & \dots & x_d \end{bmatrix} \end{cases}$$

The objective is to $learn$ the weights $\boldsymbol{w}$ such that correct outputs are generated given the inputs $\boldsymbol{x}$. If input is fed straight from the environment, a perceptron can be used to implement a linear fit; therefore it lies in the category of linear classifiers [1].

B. Cross-validation

Training and testing a certain classifier model using the same data is a methodological mistake: a model that would just repeat the labels of previously seen samples will have a perfect score but fail to predict anything useful on yet unseen data. This situation is called **overfitting**.

To avoid it, it is a common practice to hold out a part of the available data as a **test set**. Nonetheless, when evaluating different settings (*hyper-parameters*) for classifiers, such as the k value that must be manually set for the k-nn classifier, there is still a risk of overfitting on the test because the parameters can be tweaked until the classifier performs optimally.

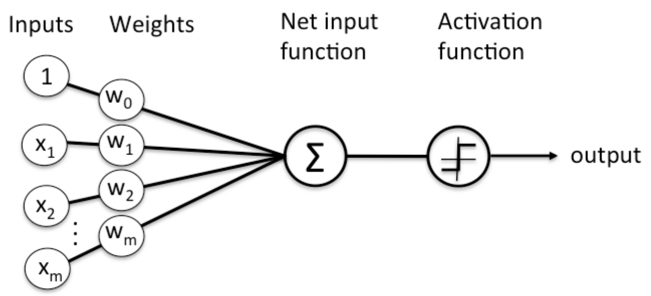


Fig. 1: Simple perceptron model

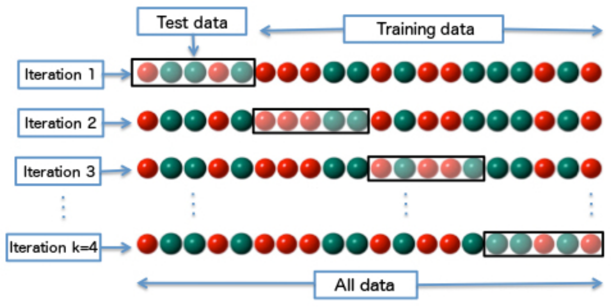


Fig. 2: k-fold cross validation diagram with $k = 4$

To solve this problem, yet another part of the dataset can be held out as a so-called **validation set**. However, partitioning data into 3 sets drastically reduces the number of samples which can be used in the learning phase and results can depend on the random splitting of the sets.

A solution to this last problem is a procedure called **cross-validation** (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, the training set is split into k smaller sets and the following procedure is followed for each of the k *folds*:

- A model is trained using $k-1$ of the folds as training data.
- The remaining data is used as a test set to compute performance metrics (model validating).

The performance metric(s) reported by k-fold CV can then be averaged to obtain a more robust validation. This approach can be computationally expensive, but does not waste too much data [2].

II. Approach

In this section we describe the approach taken for the training (learning) and the prediction phase of the perceptron.

A. Training step (learning)

The neural network perceptron is a way of representing a hyperplane via its weight values. Given a dataset, there are two main ways to compute the weight values: **offline** and **online** learning. Offline learning is used when we're given the whole dataset, whereas online learning is used when we are given observations one by one and we would like the network to update its parameters after each observation, adapting itself slowly in time.

1) Online learning: This approach is interesting for a number of reasons:

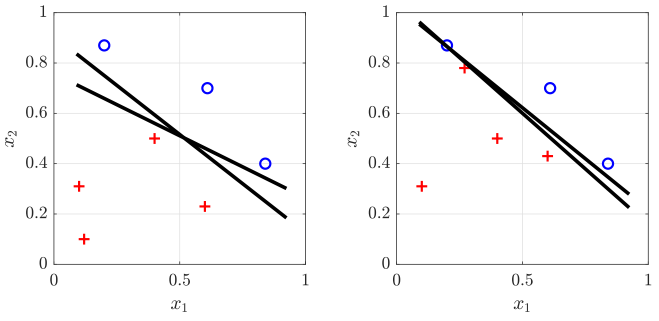
- It saves us the cost of storing training data and optimization intermediate results in an external memory.
- Simpler approach where we do not require to solve a complex optimization problem over a *big* dataset.
- Problem may change in time, meaning that data distribution is not fixed, and training set cannot be chosen a priori. For example, a speech recognition system that adapts itself to its user.

In online learning we start from random initial weights and at each iteration we adjust the parameters in order to minimize the error between the desired and the predicted output.

For this implementation of the perceptron we'll use the *sign* function as the activation function. Algorithm 1 describes the steps to follow when performing binary classification, where η is the *learning rate*, l is used a subscript to denote a single observation and m is the total number of observations in the available training set. An *epoch* is a measure of the number of times all of the training data is used once to update the weights.

Algorithm 1 Perceptron training algorithm for binary classification using *sign* as activation function.

```
Ensure: Initialize $w$ with small random values
  $l \leftarrow 1$                                  ▷ observation index
  epochs $\leftarrow 1$
  repeat
    $a \leftarrow \text{sign}(x_l \cdot w)$
    $\delta \leftarrow 0.5 \cdot (y_l - a)$                        ▷ error
    $\Delta w \leftarrow \eta \cdot \delta \cdot x_l$
    $w \leftarrow w + \Delta w$                           ▷ update weights
    $l \leftarrow l + 1$                                ▷ one by one
    if $l > m$ then
      epochs $\leftarrow$ epochs $+1$
      if no errors then                  ▷ data linearly separable
        break loop
      end if
      $l \leftarrow 1$                                 ▷ reset index
    end if
  until epochs $\geq$ maxEpochs
```



(a) dataset 1 **(b)** dataset 2

Fig. 3: Different linear boundaries found using the perceptron.

As previously mentioned, the perceptron is a linear classifier, therefore it will never reach the state where all inputs are classified correctly if the training set is not linearly separable, i.e. if positive examples cannot be separated from negative ones by a hyperplane. In case of having *non* linearly separable data, the learning will fail completely using this standard algorithm since it provides no *approximate* solution. But if the training set *is* linearly separable, the perceptron is guaranteed to converge.

B. Testing step (prediction)

The prediction phase for the perceptron is straightforward since by now a hyperplane that separates the positive from the negative class has been found. Unseen data is send through the inputs of the perceptron and the output will tell us if the given observation correspond to the positive or negative class. Mathematically, the prediction is carried out as follows (assuming *sign* is the activation function).

$$\hat{y} = \text{sign}(\boldsymbol{w}^T \, \boldsymbol{x}) \tag{2}$$

III. Experimental Results & Analysis

A. Perceptron classifier

A MATLAB class `perceptronKlassifier` was developed in MATLAB following the algorithm presented in the previous section. We can perform the training and prediction phase using the following snippet.

```
% create classifier object
percep = perceptronKlassifier();

% adjust hyper-parameters
percep.lr = 0.5;             % learning rate
percep.max_epochs = 200;     % 1000 by default

% train classifier
percep.learn(train.X, train.t);

% access learning results locally stored in...
percep.w                 % weight vector
percep.total_epochs      % epochs needed for training

% get predicted output and compute its accuracy
y_pred = percep.predict(test.X);
acc = mean( test.t == y_pred ); % accuracy
```

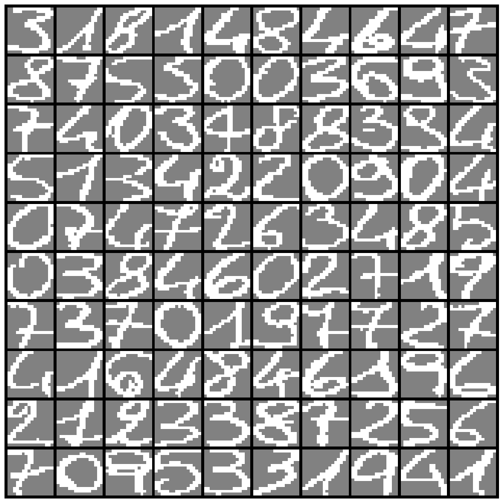


Fig. 4: Randomly picked samples of the semeion-digits dataset.

TABLE I: Epochs needed to linearly separate each class (one-vs-all)

Digit	0	1	2	3	4	5	6	7	8	9
Epochs	10	43	22	32	24	21	21	38	187	38

1) 2D visualization: In order to verify that our perceptron works as expected, we train it using two simple 2D datasets and plot the resulting hyperplane. Although results shown in figure 3 indicate that the perceptron does converge on *some* solution, it has no way of choosing which one is the *best*, which will impact classification performance when using unseen data. The perceptron of optimal stability, nowadays known as the linear support vector machine was designed to address this problem but it is out of the scope of this report.

2) Is the semeion-digits dataset linearly separable?: Since perceptrons work optimally only with linearly separable data, we should ask ourselves: Is the semeion-digits data linearly separable? Then answer is *yes* and the solution was found by training 10 perceptrons using the one-vs-all approach. Each one of them succeeded in finding a linear boundary to separate the given binary classes in a finite number of epochs (table I).

B. Cross-validation

The MATLAB function `cross_val_score` was created in order to simplify code structure. It takes as inputs a classifier object, the training set and the number of folds. Outputs the scores in a *struct* containing the performance metrics for each fold, the k trained classifiers and the k chunks of the split data. A simple snippet is provided to show its easy use.

```
knn = kNNKlassifier(5);     % k=5
percep = perceptronKlassifier();

[train, ~] = stratified_split(X, t, 1);

kFolds = 5;
[knn_scores, ~] = cross_val_score(knn, ...
    train.X, train.t, kFolds);
[percep_scores, ~] = cross_val_score(percep, ...
    train.X, train.t, kFolds);
```

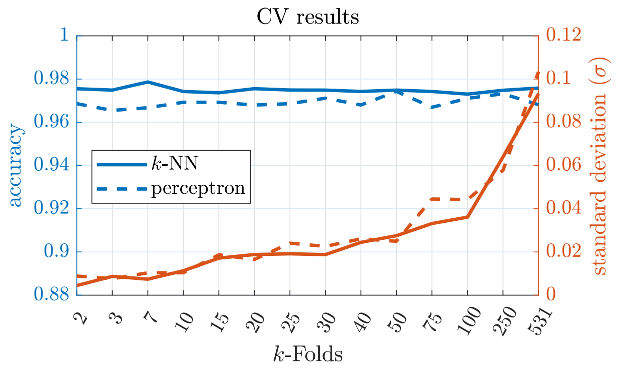


Fig. 5: Cross-validation results varying number of folds. Note that the left axis corresponds to the accuracy and the right axis to the standard deviation

TABLE II: LOOCV results using whole dataset

	accuracy	standard deviation
k-NN	0.9743	0.1584
perceptron	0.8983	0.3023

1) How does the number of folds influence the classifier performance?: To get a better understanding on how the number of folds impacts on the classifier performance, we performed the following test:

1) Use the whole dataset as training set
2) Binarize target vector using digit *1* as target class.
3) Fix hyper-parameters, if any, of the classifiers to be tested.
4) Perform CV with the k-NN and perceptron classifiers increasing the number of folds progressively.
5) Plot performance results in the form of box plots, which is a compact way of graphically representing data distribution.

Cross-validation wasn't designed for improving accuracy itself but rather to obtain a general model to use on unseen data. For choosing a good value of k we must minimize *variance* and *bias*. When having a large number of folds we're left out with less observations for testing, thus predictions are more sensitive to misclassification and, in turn, we get higher variance as shown in figure 5. A small number of folds isn't the solution either since we don't want to hold-out too much of the data (high bias) for testing. It seems that keeping the number of folds around 10 is a good variance-bias trade-off while ensuring that the classifier itself has low bias and variance.

2) Leave-one-out cross-validation: LOOCV can be seen as a special case of k-fold CV when k equals the total number of observations in the training set. This approach requires excessive computation and provides the most variance since, at each fold, accuracy can only be either 100% or 0%. Nonetheless, this turns out to be handy for analyzing the percentage of outliers or hard samples in the dataset. From table II, the outlier percentage can be obtained from the complement of accuracy (error percentage) where k-NN outperforms the perceptron.

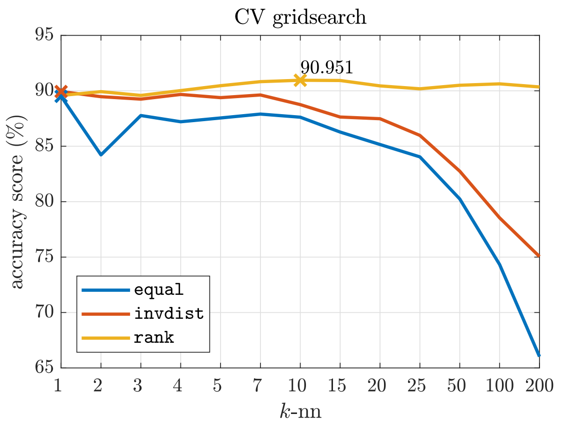
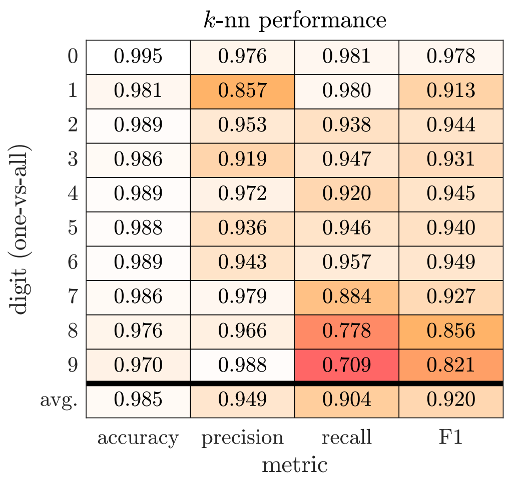


Fig. 6: k-NN average accuracy results using 5x5CV grid search.

TABLE III: k-NN average performance scores (5x10CV)

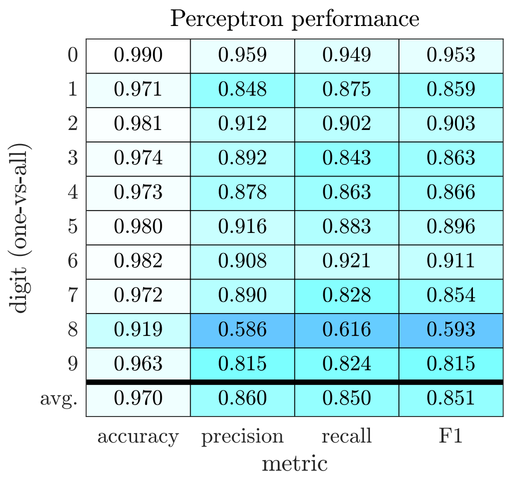


k-nn performance

digit (one-vs-all)	accuracy	precision	recall	F1
0	0.995	0.976	0.981	0.978
1	0.981	0.857	0.980	0.913
2	0.989	0.953	0.938	0.944
3	0.986	0.919	0.947	0.931
4	0.989	0.972	0.920	0.945
5	0.988	0.936	0.946	0.940
6	0.989	0.943	0.957	0.949
7	0.986	0.979	0.884	0.927
8	0.976	0.966	0.778	0.856
9	0.970	0.988	0.709	0.821
avg.	0.985	0.949	0.904	0.920

metric

TABLE IV: Perceptron average performance scores (5x10CV)



Perceptron performance

digit (one-vs-all)	accuracy	precision	recall	F1
0	0.990	0.959	0.949	0.953
1	0.971	0.848	0.875	0.859
2	0.981	0.912	0.902	0.903
3	0.974	0.892	0.843	0.863
4	0.973	0.878	0.863	0.866
5	0.980	0.916	0.883	0.896
6	0.982	0.908	0.921	0.911
7	0.972	0.890	0.828	0.854
8	0.919	0.586	0.616	0.593
9	0.963	0.815	0.824	0.815
avg.	0.970	0.860	0.850	0.851

metric

3) Adjusting hyper-parameters with CV: Cross-validation was initially designed as a method to train a classifier such that we end up with a generalized model before it's let out into the world. This generalization is achieved by fine-tuning the classifier's hyper-parameters.

Since the standard perceptron algorithm aforementioned has no hyper-parameters there is not much we can do to enhance its performance when testing it with unseen data. On the other hand, the implemented k-NN algorithm has the following hyper-parameters:

- `k`: number of k-nearest neighbors used to classify unseen data. Can take any value from 1 to the total number of observations in training set.
- `weighfcn`: defines the weight contribution of each neighbor.
  - `equal`: No weighting
  - `invdist`: weight is 1/distance
  - `rank`: weight is 1/rank

Using 50% of the dataset for CV tuning and 50% for blind validation, we performed an exhaustive search (also referred to as CV grid search) over specified hyper-parameter values for the k-NN classifier. Average accuracy results, using 5x5CV (5 iterations of 5-fold cross-validation) on each hyper-parameter combination, are displayed in figure 6. Accuracy tends to decrease when taking into account a large number of neighbors except when using `rank` for weighting contributions. The hyper-parameters [k, `weighfcn`] that yield the highest average accuracy of 90.951% is [10, `rank`]. Alternative performance metrics show the same behavior:

- Highest average precision score: 0.916 $\leftarrow$ [10, `rank`]
- Highest average recall score: 0.909 $\leftarrow$ [10, `rank`]
- Highest average F1 score: 0.908 $\leftarrow$ [10, `rank`]

We'll therefore choose these hyper-parameters for further testings and comparisons.

C. Perceptron vs k-NN algorithm

So far we have two classification models and we would like to compare their results. These classifiers are quite different from one another, the main difference being their training and testing phase. The perceptron carries out most of the heavy computing at the training phase, contrary to the k-NN algorithm (lazy-learner) where all the computation is left to the predicting phase. We're particularly interested in knowing how well our classifiers perform on predicting true positives, therefore we'll mainly use *recall*, *precision* and *F1* scores to measure and compare their performance.

1) Binarized targets: Both classifiers are able to predict binary outputs, which is the simplest case of a classification problem. In order to obtain results as neutral as possible, we used all the available data and performed 5x10CV. Results then were averaged and are presented in tables III and IV. Each row is independent from one another since for each digit we performed a one-vs-all binarization.

In the k-NN case, precision and recall show some mayor differences between one another. Taking digit 9 as an example, we get a small number of false positives (high precision) but we have a large number of false negatives (low recall); in other words, whenever k-NN predicts a 9 we can be pretty confident the number is an actual 9, however, when the prediction is *not a* 9 there is 30% chance the prediction is mistaken. Digit 1 presents an inverse behavior; we're confident when prediction is *not a* 1, but other digits may be falsely classified as 1.

Focusing now on the perceptron performance, precision and recall present a similar score for all binary outputs. Meaning that regardless of the prediction output, we'll always have some uncertainty on it; specially for digit 8 where the chance of getting a wrong prediction is 40%.

Both classifiers show high average accuracy, but when referring to other metrics the panorama changes. Unbalanced class proportion is the main reason for this behavior. The F1 column can be seen as a weighted average of precision and recall, giving us a good picture of the overall performance.

2) Multiclass classification: Is there a way to know which pair of digits get confused with one another? Previously, we had no way of knowing if the classifier was confusing two of the 10 available classes. With multiclass classification we're able to predict between more than two categories $y = \{0, 1, ...n\}$. Then, we can use this prediction to plot a *confusion matrix*, which is a powerful tool for visualizing algorithm's performance. The name stems from the fact that it makes it easy to see if the system is confusing two classes.

The k-NN algorithm requires no modification since it handles both binray and multiclass classification in the same way.

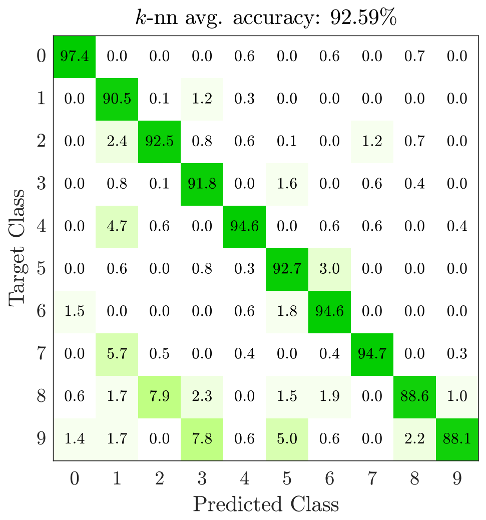
The perceptron is by nature a binary classifier, therefore we divide our problem into $n+1$ binary classification problems; in each one, we predict the probability that y is a member of one of n classes and then use the perceptron that returned the highest value as our prediction [3]. In our case, the use of *sign* as an activation function doesn't output a probability, instead we'll take the weighted sum to compare and choose the highest.

$$
\begin{gathered}
y \in \{0, 1, ...9\} \\
h^{(0)}(x) = \boldsymbol{w}^{(0)}\boldsymbol{x} \\
h^{(1)}(x) = \boldsymbol{w}^{(1)}\boldsymbol{x} \\
\vdots \\
h^{(9)}(x) = \boldsymbol{w}^{(9)}\boldsymbol{x} \\
\text{prediction} = \max_i(h^{(i)}(x))
\end{gathered}
$$

Once more, we performed 5x10CV and computed a confusion matrix at each step. We then averaged all 50 of them and applied F1 scoring over all the elements as shown in figure 7. One of the first things we notice when looking at the results is that the confusion matrix for the perceptron is *dirtier* in the sense that there are lots of weak misclassifications, whereas the k-NN has few but strong confusions between classes.

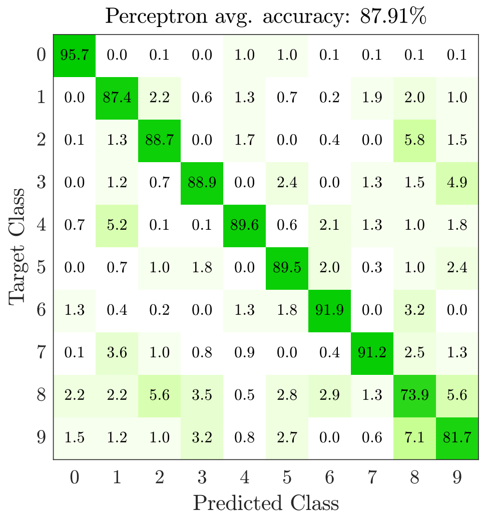
Addressing the overall performance of each classifier, we can point out the following:

- k-NN classifier
  - Strong confusion between $8 \leftarrow 2$ and $9 \leftarrow 3$.
  - Precision avg. score: 0.933
  - Recall avg. score: 0.925.
  - F1 avg. score: 0.926.
- Perceptron multiclass classifier
  - Strong confusion between $8 \leftrightarrow 2$, $9 \leftrightarrow 8 \leftarrow 1$.
  - Precision avg. score: 0.884
  - Recall avg. score: 0.879.
  - F1 avg. score: 0.878.
  - Digit 8 has a poor score and, by looking at table I, we note that the perceptron needs a large number of epochs to find a suitable hyperplane, suggesting that this high dimensional boundary is in a *tight* space (prone to misclassification). On a 2D scenario it would look similar to figure 3b.



Target \ Predicted	0	1	2	3	4	5	6	7	8	9
0	97.4	0.0	0.0	0.0	0.6	0.0	0.6	0.0	0.7	0.0
1	0.0	90.5	0.1	1.2	0.3	0.0	0.0	0.0	0.0	0.0
2	0.0	2.4	92.5	0.8	0.6	0.1	0.0	1.2	0.7	0.0
3	0.0	0.8	0.1	91.8	0.0	1.6	0.0	0.6	0.4	0.0
4	0.0	4.7	0.6	0.0	94.6	0.0	0.6	0.6	0.0	0.4
5	0.0	0.6	0.0	0.8	0.3	92.7	3.0	0.0	0.0	0.0
6	1.5	0.0	0.0	0.0	0.6	1.8	94.6	0.0	0.0	0.0
7	0.0	5.7	0.5	0.0	0.4	0.0	0.4	94.7	0.0	0.3
8	0.6	1.7	7.9	2.3	0.0	1.5	1.9	0.0	88.6	1.0
9	1.4	1.7	0.0	7.8	0.6	5.0	0.6	0.0	2.2	88.1

(a) k-NN performance



Target \ Predicted	0	1	2	3	4	5	6	7	8	9
0	95.7	0.0	0.1	0.0	1.0	1.0	0.1	0.1	0.1	0.1
1	0.0	87.4	2.2	0.6	1.3	0.7	0.2	1.9	2.0	1.0
2	0.1	1.3	88.7	0.0	1.7	0.0	0.4	0.0	5.8	1.5
3	0.0	1.2	0.7	88.9	0.0	2.4	0.0	1.3	1.5	4.9
4	0.7	5.2	0.1	0.1	89.6	0.6	2.1	1.3	1.0	1.8
5	0.0	0.7	1.0	1.8	0.0	89.5	2.0	0.3	1.0	2.4
6	1.3	0.4	0.2	0.0	1.3	1.8	91.9	0.0	3.2	0.0
7	0.1	3.6	1.0	0.8	0.9	0.0	0.4	91.2	2.5	1.3
8	2.2	2.2	5.6	3.5	0.5	2.8	2.9	1.3	73.9	5.6
9	1.5	1.2	1.0	3.2	0.8	2.7	0.0	0.6	7.1	81.7

(b) Perceptron performance

Fig. 7: Classifiers' confusion matrices with F1 scores.

IV. Conclusions

Cross-validation is a powerful tool to fine-tune hyper-parameters in order to obtained a generalized classifier. While there are multiple approaches to CV, using k-fold CV proves more versatility and the value k should picked based on minimizing *bias* and *variance*. Additionally, CV can be used to better assess classifiers based on their performance regardless of how the data is used for training since.

The traditional perceptron algorithm presented in this report can be implemented easily but it has an important limitation. Data must be linearly separable between all classes, otherwise the perceptron won't be able to converge into a solution and doesn't provide any *approximation*. However, it provides some advantages such as low-memory usage and the ability to adapt to changing problems.

Comparing k-NN and perceptron classifiers, we assert the following:

- Computationally speaking, the k-NN's complexity depends on the size of the training data, and the perceptron's complexity relies on the number of classes in the target vector.
- Perceptron's predictions can vary a lot due to the **random** initialization of weights and the fact that these are not *optimal*, whereas predictions obtained with k-NN are more stable since they do not rely on any random process and are only affected by the way we define the weighted contribution of each neighbor.
- On the tests conducted, k-NN outperformed the perceptron in all cases

The perceptron algorithm has still room for improvement, we could use other activations functions such as *sigmoid* or *ReLU*. These in turn require some more complicated cost functions but could lead to better results. As proved in the results section, semeion-digits are all linearly separable from one another and we can improve our prediction by finding the optimal linear boundary between two classes, which nowadays can be accomplished with the support vector machine (SVM).

REFERENCES

[1] E. Alpaydm, *Introduction to Machine Learning*. The MIT Press, 2010.
[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[3] A. Ng, "Machine learning," https://www.coursera.org/learn/machine-learning, 2012, accessed 10/11/17.