university of groningen

The Effect of Regularization on the Performance of K-Nearest Neighbours and Linear Regression in Handwritten Digit Recognition

Group 01
Ivo de Jong (s3174034)
Travis Hammond (s2880024)
Elisa Oostwal (s2468255)
Hayo Ottens (s2197839)

February 10, 2020

Abstract

This research will model the recognition of handwritten digits using the K-nearest-neighbours (KNN) algorithm. The algorithm will be applied to the data-set from J. Kittler et al. [4]. The performance of the algorithm will be compared to the linear regression classifier as our baseline. For both classifiers, pre-processing and validation steps will be identical. Model validation will be done by K-fold cross validation. To expand the training data, noisy duplicates will be created by adding random Gaussian distributed values to each pixel. The main goals of the research will be:

- identifying the relation between the amount of noise added to each pixel and the performance of the algorithms;
- finding the optimal number of neighbours K used in the KNN algorithm;
- identifying the relation between the added noise and the optimal number of neighbours K in the KNN algorithm.

1 Introduction

The main aim of Machine Learning is to find a model that best describes the origin of the data. Given some data, retrieving the underlying mechanism is a challenge. On the contrary, it is relatively easy to make a classifier perform well on specific training data: using for example a large neural network enables to memorize all training samples. This however results in *overfitting*: the classifier is tailored to make good prediction on the data which it has seen, but performs poorly on data which it has not been presented with. To prevent this, it is key to design the classifier in such a way that it is general enough to perform well even on unseen data, but also be specific enough to cover the mechanism hidden in the data. The parameters which we can tune in order to optimize the performance of the machine learning algorithm are called *hyperparameters*. Our goal is thus to find the optimal values of the hyperparameters.

In this paper, we apply some of the renowned methods to construct two classifiers. Specifically, we use a *Linear Regression* (LR) classifier as a baseline and compare its performance to a *K-Nearest Neighbours* (KNN) classifier. Moreover, we will explore regularization techniques in order to improve the performance. We apply dimension reduction by PCA (Section 1.4) and regularization by adding noise (Section 1.5). We train the classifiers on the Digits dataset by Kittler et al. [4] and subsequently use K-fold cross validation in order to estimate their risk, i.e. their performance on unseen data. The performances of the models are then compared.

1.1 Digits Dataset

The Digits dataset taken from [4] is used to test the influence of the hyperparameters on the performance of the classifiers. This dataset consists of two-thousand gray-scale 15×16 pixel images of handwritten Hindu-Arabic numerals: the digits $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Every $d \in D$ occurs two-hundred times, so every digit is represented equally, meaning it is a balanced dataset. The dataset is sorted such that the first two-hundred vectors of dimension $15 \times 16 = 240$ are of class 0, the next two-hundred of class 1 and so on. In this dataset, the gray-scale is mapped to integers in the range $[0, 6]$, where 0 denotes white and 6 denotes black.

1.1.1 One-hot encoding

We use one-hot encoding in order to label the examples [3]. One-hot encoding offers a way to transform the class labels to $||D||$-dimensional vectors, where $||D||$ the number of classes, this case $||D|| = 10$. The resulting vector will be zero everywhere, except on the index of the represented number. For example, the label '0' will be represented as $y_0 = \big[1, 0, 0, 0, 0, 0, 0, 0, 0, 0\big]'$.

1.2 Classification methods

As mentioned before, we use two different classifiers in order to be able to draw conclusions about the relation between regularization and model performance. The two

classifiers we will be using are Linear Regression and K-Nearest Neighbours. They are explained in the following subsections.

1.2.1 Linear Regression

As our baseline we use Linear Regression. This classifier aims to find a weight vector which transforms the input vectors such that it differs minimally with the target output [6].

Data A set $(\mathbf{x_i}, y_i)_{i=1,\ldots,N}$ of n-dimensional input vectors $\mathbf{x_i}$ and scalar targets y_i.

Wanted An n-dimensional row weight vector $\mathbf{w}$ which solves the linear regression objective:

$$w = \underset{\mathbf{w}^*}{\operatorname{argmin}} \sum_{i=1}^{N} (\mathbf{w}^* \mathbf{x}_i - y_i)^2$$

Procedure

Step 1 Sort the input vectors as columns into an $n \times N$ matrix X and the targets into an N-dimensional vector y.

Step 2 Compute the (transpose of the) result by

$$w' = (XX^{-1})Xy$$

1.2.2 K-Nearest Neighbours (KNN)

Another classifier we will be studying is the K-Nearest Neighbours (KNN) algorithm. KNN is an algorithm of the 'lazy learning' type, since no actual training needs to be done [9]. For every unclassified example X_{new} the algorithm calculates the distance between the example and the other already classified datapoints $\{X, Y\}$, where $X \in \mathbb{R}^d$ the datapoint and $Y \in \mathbb{R}^d$ the label. The K nearest datapoints are selected, and a majority vote is then used to determine the class Y_{expected} of X_{new}.

Data A set $(\mathbf{x_i}, y_i)_{i=1,\ldots,N}$ of n-dimensional input vectors $\mathbf{x_i}$, scalar targets y_i, an unclassified example X_{new} and a desired K.

Wanted A label for X_{new}

Procedure

Step 1 Calculate the distances between X_{new} and all $x_i, i \in N$ by some distance measure (e.g. Euclidean)

Step 2 Make a frequency table of the labels of the K examples with lowest distance and output the most occurring label

1.3 K-Fold Cross Validation

A good classifier should have the right amount of flexibility. If the model is too tailored on the data used during training, the classifier may perform badly on new data (*overfitting*). On the other hand, it may be too general not and fully convey the information which is present in the data (*underfitting*). We therefore want to get an idea of the classifier's performance on data which is not presented during training. To this end, we use K-fold cross validation [5]. In this method, the training data is partitioned in k batches or *folds* of equal size. One of the folds is used as testing data and the remaining $(k-1)$ folds are used as the training data. The performance of the classifier is computed as the average testing error over the k results. In our experiment we set $k = 10$.

1.4 Principal Component Analysis

One of the hyperparameters that can be configured is the number of dimensions of the data. Reducing the used dimensions of the dataset also reduces the training time and storage. Moreover, removing dimensions that offer little to no information may improve the actual interpretation of the parameters of the machine learning model and avoids the curse of dimensionality. In order to find the most meaningful dimensions we apply Principal Component Analysis (PCA). Effectively, PCA orders the feature vectors of the data from most to least meaningful [8]. Once this analysis has been performed, the optimal number of features is determined by performing cross validation across classifiers that use a different number of features.

PCA performs the following steps:

Data A set $(\mathbf{x}_i)_{i=1,\ldots,N}$ of n-dimensional pattern vectors.

Result An n-dimensional mean pattern vector μ and m principal component vectors arranged column-wise in an $n \times m$ sized matrix U_m.

Procedure

Step 1. Compute the pattern mean μ and center the patterns to obtain a centered pattern matrix $\bar{X} = [\bar{x}_1, ..., \bar{x}_N]$.

Step 2. Compute the Singular Value Decomposition (SVD) $U\Sigma U'$ of $C = 1/N\bar{X}\bar{X}'$ and keep from U only the first m columns, making for a $n \times m$ sized matrix U_m.

Compression In order to compress a new n-dimensional pattern to a m dimensional feature vector $\mathbf{f}(\mathbf{x})$, compute $\mathbf{f}(\mathbf{x}) = U'_m\bar{\mathbf{x}}$.

Decompression In order to approximately restore $\mathbf{x}$ from its feature vector $\mathbf{f}(\mathbf{x})$, compute $\mathbf{x}_{\text{restore}} = \mu + U_m\mathbf{f}(\mathbf{x})$

1.5 Regularization

Another technique used to prevent overfitting is *regularization* [1]. Regularization is a means to generalization. Regularization is applied to prevent the classifier from "mem-

orizing" specifics unique to the training dataset but which are not useful for generalizing to new data. Most regularization is done either done by limiting model complexity, or by adding noise to the training data in order to force the model to ignore the specifics and learn underlying patterns. The previously mentioned PCA can also be regarded as a regularization technique, since it forces the model to ignore certain details.

In LR regularization can be achieved by using ridge regression [7]. Here a term $\alpha I_{n\times n}$ is included which regularizes the size of the weights:

$$W'_{\text{opt}} = (XX' + \alpha I_{n\times n})^{-1} Xy$$

Larger values of α are associated with smaller weights, whereas large weights usually indicate overfitting. In Section 1.6 we can see the effect of ridge regression for $\alpha = 10$ as opposed to $\alpha = 0$, which means that no regularization is applied.

However, ridge regression is a regularization technique which does not work for KNN. Therefore, we use another technique to apply regularization, namely the addition of Gaussian noise [10]. For every pixel of an example we add noise as drawn from a Gaussian distribution with zero mean μ and a given variance σ. In this way we can make multiple copies of the same data with different noise. The main idea is that by adding some distortion, the model can rely less on certain features. Eventually, the weights for these features will decrease, which makes it less prone to overfitting. We use this regularization technique since it can be applied on both KNN and LR. In this paper we investigate the effects of adding Gaussian noise to the dataset, where we only vary the variance of the Gaussian.

1.6 Preliminary tests and Expectations

We performed a preliminary test in which we varied the number of principal components to study its effect on the Mean Squared Error (MSE) of the Linear Regression classifier. From the results shown in Figure 1 we infer that the lowest test error is obtained at 30 principal components. When more principal components are being considered, the MSE on unseen test data increases. This means that the model is overfitted during the training.

In another preliminary test to observe the effects of regularization, we set $\alpha = 10$ in the Ridge Regression formula, as seen in Section 1.2.1. From its results shown in Figure 2, it can be observed that the Mean Squared test error barely increases anymore even when more principal components are being considered. From this it is evident that regularization indeed limits overfitting.

From these preliminary results, we expect that all models perform best with about 30 principal components considered. Moreover, when Gaussian noise is added, also models with more principal components will perform well. However, we believe that when too much noise is added to the model, the performance decreases.

It can be expected that the performance for KNN will be better than that of LR. Previous experiments showed that KNN is was very effective at recognizing handwritten digits on the MNIST dataset [2]. It can be expected that performance will form a bow shape for each parameter. That is, for example, with $K = 1$ the KNN algorithm won't be able to take advantage of the collective intelligence that forms from multiple

datapoints. However, with $K = 100$ one can be certain that the datapoints that are considered are no-longer relevant for the entry that is to be classified. A similar effect should exist for the number of principle components m. There may be ideas that certain interactions between the parameters exist, but is challenging to find a well-argue hypothesis to match those ideas. Instead any possible interactions should be inferred from the results.

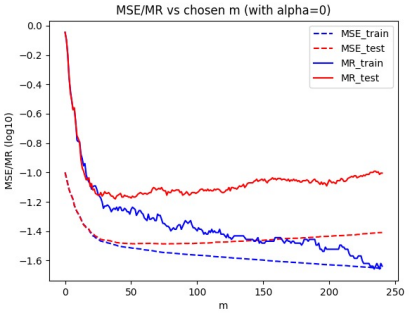

Figure 1: MSE of Ridge Regression with $\alpha = 0$

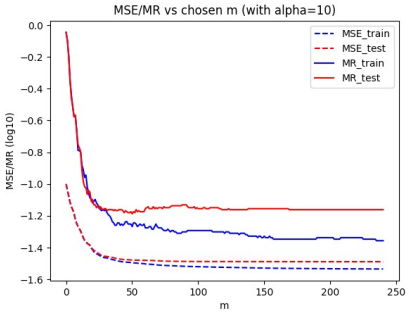

Figure 2: MSE of Ridge Regression with $\alpha = 10$

2 Method

In this section we describe how we have implemented the methods described in section 1.2. We also elaborate on how we intend to optimize the hyperparameters.

2.1 Implementation

Our project is implemented with Python 3.6, mainly making use of the `numpy` and `sklearn` packages. The code can be found in the Appendix A.

We start off by loading the data and creating the labels. Subsequently, we specify the values for the hyperparameter settings: the number of principal components, the amount of noise, and the number of nearest neighbours. The settings for these hyperparameters are described in Section 2.2. We then create two separate pipelines: one for Linear Regression and one for KNN. In both pipelines the data is first centered, after which PCA is applied. After the pre-processing several models, each with different hyperparameter settings, are trained and tested for performance. This can be done simultaneously for multiple models by means of multi-threading. For every model the pipeline is run in combination with 10-fold cross-validation. Finally, the results are saved in a Pickle file.

2.2 Hyperparameter settings

We vary the hyperparameters of the different methods. Our goal is to find the combination of values which gives the optimal result (lowest test error). We choose to test for the following values:

- Number of principle components $m \in \mathbb{N}, [20, 50]$
- Amount of noise $\sigma \in \mathbb{R}, [0, 3]$ in steps of 0.25.
- Number of nearest neighbours $K \in \mathbb{N}_{\text{odd}}, [3, 9]$
- Number of training-test data splits for K-Fold Cross-Validation $k = 10$

The noise will be added to ten copies of the original data. The training data then consists of eleven copies of each datapoint: the original datapoint and ten copies with additional noise. In the validation step only the original data without noise that was not part of the training will be used. This ensures that the classifier is validated on the actual task of digit classification, and not on the task of noisy digit classification.

2.3 Performance

The performance measures used are the mean misclassification rate (MR), precision (prec), recall (rec), accuracy (acc) and F1-score (F1). These scores are measured by computing the average over the ten folds of cross-validation.

3 Results

We have acquired the performance measures mentioned in section 2.3 as a function of Gaussian noise variance (σ) and number of principal components (m) for both the Linear Regression and K-Nearest Neighbour classifier. Of these performance measures only the mean Misclassification Rate (MR) is presented in the following subsections, as the other performance measures are in accordance.

3.1 Linear Regression

Since we use Linear Regression as a baseline, we cover its results first. In Figure 3 we present a plot of the Misclassification Rate as a function of the number of principal components m and the variance of the noise added σ. It becomes evident that the MR decreases when m increases. Figure 5 shows this effect as well. From Figure 3 we do not observe an effect of the noise on the MR. Figure 4 confirms that there is little to no effect of the noise on the performance of the LR classifier.

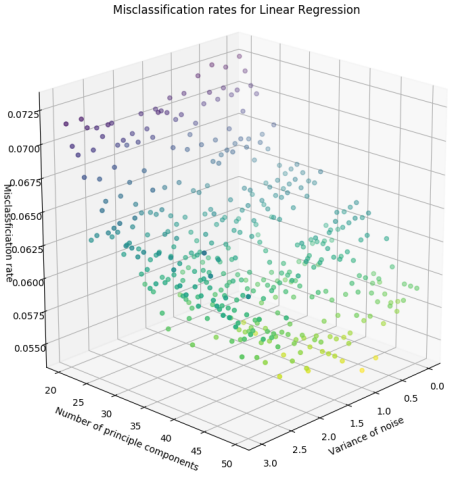

Figure 3: 3D plot of mean misclassification rate as a function of number of principal components m and variance of noise σ.

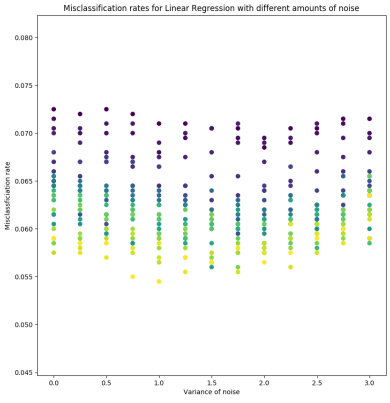

Figure 4: Misclassification rates for Linear Regression as a function of variance of noise σ.

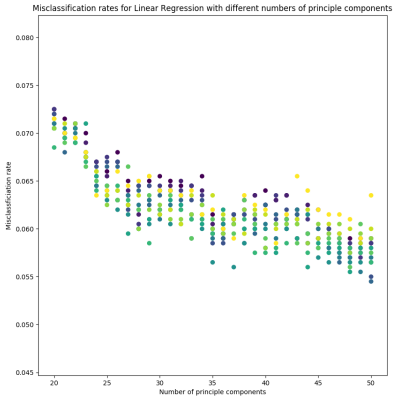

Figure 5: Misclassification rates for Linear Regression as a function of the number of principal components m.

3.2 K-Nearest Neighbours

The misclassification rates of the KNN classifier are visualized for $K \in \{3, 5, 7, 9\}$ in Figure 6. From the plot we infer that the misclassification rate decreases as both the noise and number of neighbours increases.

Although this plot gives some insight, little can be said about the performance in general. Therefore, we have fitted the individual datapoints to quadratic planes. Figure 7 shows the MR planes for different values of K. Out of the four planes we observe the performance is best for $K = 9$. Furthermore, from the plot a clear trend becomes apparent: for all values of K we observe that the MR is negatively correlated with both σ and m. Moreover, the effect of both σ and m are stronger for larger values of K. Interestingly, it appears that individually optimizing each hyperparameter gives a better performance than optimizing them simultaneously; the perfect set of parameters seems to converge to a single point, where the optimization of any single parameter does not seem to have a notable interaction with the optimization of another.

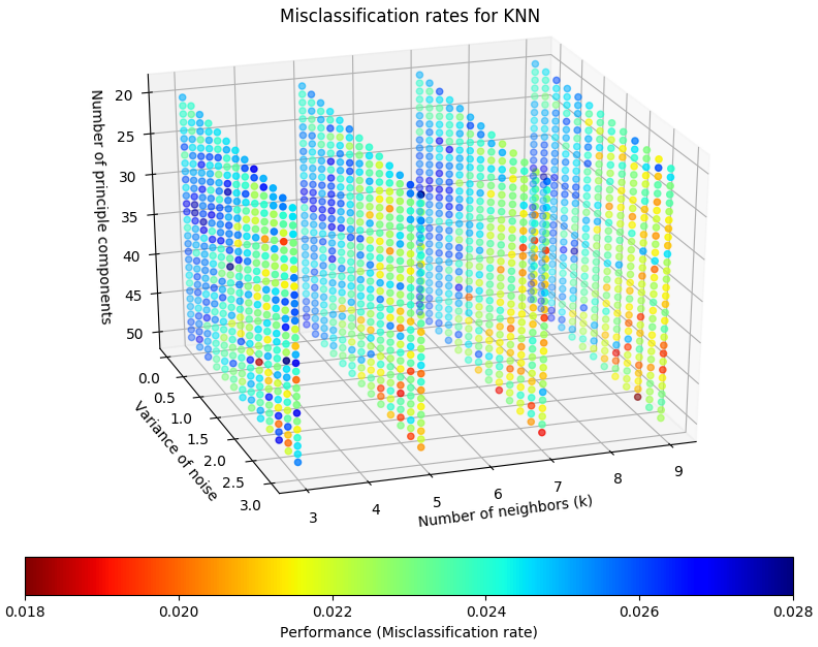

Figure 6: Plot of mean misclassification rate as a function of the number of principal components m and noise variance σ.

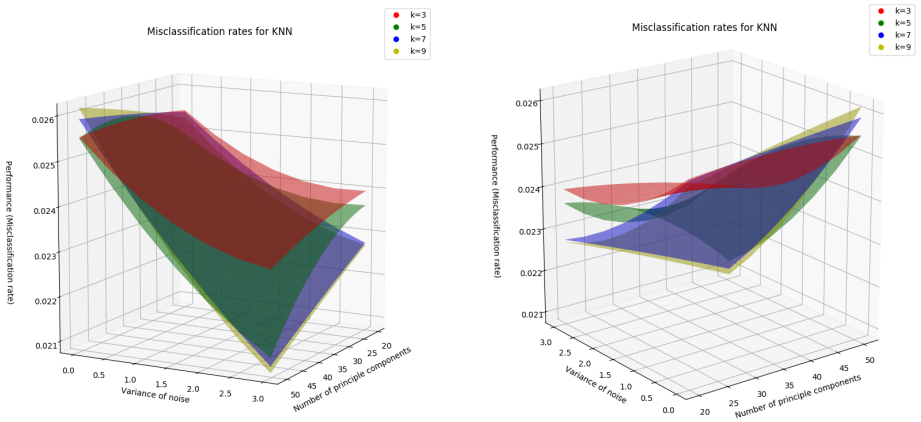

Figure 7: Quadratic Misclassification Rate planes for $K \in \{3, 5, 7, 9\}$.

3.3 LR vs KNN

Figure 8 shows a comparison between the performance of the Linear Regression classifier and the K-Nearest Neighbour classifier, where $K = 9$. We observe that KNN clearly performs better than LR, independent of the number of principal components and amount of added noise. Moreover, we note that the performance of the LR classifier is more dependent on m and σ compared to the KNN. From the curvature of the performance planes we conclude that there is an ideal noise spread between $[1.0, 2.0]$.

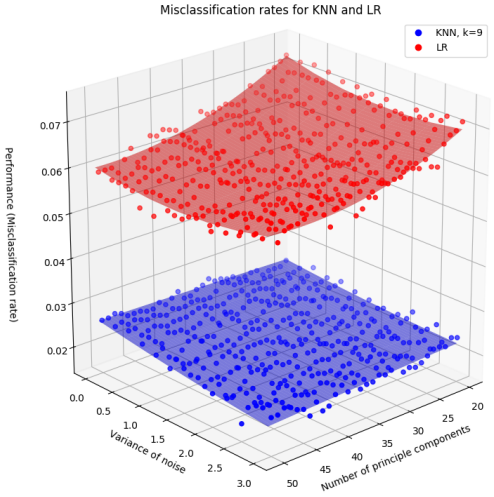

Figure 8: Missclassification rates as a function of Gaussian noise variance σ and number of principal components m for KNN and LR.

4 Discussion

For both the Linear Regression classifier and the K-Nearest Neighbour algorithm the performance increases as the number of dimensions is reduced. During our preliminary tests, we expected the optimal value for the number of dimensions to be around 30. However, from the results it appears that the best performance is found when we use the largest number of dimensions used. In fact we expect the optimal number of dimensions to be outside of our tested range.

For LR there appears to be little to no effect of the amount of noise on the performance. For KNN, however, this effect is present: increasing the amount of noise has a positive effect on the performance. Moreover, this effect is larger with larger values for K. We observed the best performance of KNN for $K = 9$, but we would have to increase our range of values for K in order to conclude that this is really the optimal number of nearest neighbours.

The results seem to indicate that the parameters explored do not have a notable interaction; the optimization of any parameter will improve the performance. Unfortunately, the ideal performance seems to be outside the scope of the parameters tested. The computational cost for the experiment is very high, so expanding the grid search space further would be costly.

As Figure 6 shows, many of the experimental measurements do not add much to the insight. To minimize computational costs while maximizing the useful search space an alternative to a grid search might be explored. Specifically, the experiments that are closer to a good performance seem to offer more insight and also yield better models. Areas where changes of parameters do not affect the outcome of the experiment seem to offer less insight. A rough alternative for grid search in such experiments would be to do a first pass with a large stride over a parameter space, then identify the interesting areas from that first pass, and use those findings to perform experiments with smaller strides in the most interesting areas. For such a search a very wide parameter space may be explored, so that the search will be able to narrow down the space to the more interesting areas.

Finally, a remark can be made about the training- and validation time required. KNN requires substantially more time to obtain validation scores, because it needs to compute distances between datapoints in order to classify a datapoint, while training the LR classifier produces a model from which predictions can be made instantly.

5 Conclusion

In this paper we studied the effect of regularization on the performance of the two classifiers, namely Linear Regression and K-Nearest Neighbours. We examined regularization by means of dimension reduction and adding Gaussian noise to the Digits dataset provided by Kittler et al. [4]. For this study we trained models using different values for the hyperparameters, where the hyperparameter set consisted of number of principal components and variance of the added noise. For the KNN classifier we also used multiple values for K. We optimized the hyperparameters by means of searching for the smallest Misclassification Rate over the grid of hyperparameter values.

We conclude that the performance of Linear Regression is more dependent on the number of principal components and the noise spread compared to the K-Nearest Neighbours classifier. We observed that for LR using a larger number of principal components increases the performance, and that there is a range of noise values in between which the classifier performs optimally.

On the other hand, KNN seems to benefit linearly from increasing noise spread and is more robust to different values of m. Using a larger value for K increases the performance, but on the downside also increases model complexity. This, together with the obvious performance difference between them, seems to indicate that while LR remains a good algorithm to use as a baseline to compare to, it is far inferior to K-Nearest-Neighbors in terms of robustness to hyperparameters and general performance.

In general, more noise increases performance across both algorithms, indicating that it is a good regularization technique. Dimensionality reduction is useful for decreasing training time and there are indications that it is also an effective regularization. Within the range of parameter sweeps we were able to perform in the given time it appears higher values of m always increase performance.

References

[1] Peter Bühlmann and Sara Van De Geer. *Statistics for high-dimensional data: methods, theory and applications*. Springer Science & Business Media, 2011.

[2] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[3] David Harris and Sarah Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[4] Josef Kittler, Mohamad Hatef, Robert PW Duin, and Jiri Matas. On combining classifiers. *IEEE transactions on pattern analysis and machine intelligence*, 20(3):226–239, 1998.

[5] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[6] Tze Leung Lai, Herbert Robbins, and Ching Zong Wei. Strong consistency of least squares estimates in multiple regression. *Proceedings of the National Academy of Sciences of the United States of America*, 75(7):3034, 1978.

[7] Donald W Marquardt and Ronald D Snee. Ridge regression in practice. *The American Statistician*, 29(1):3–20, 1975.

[8] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[9] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[10] Yuval Raviv and Nathan Intrator. Bootstrapping with noise: An effective regularization technique. *Connection Science*, 8(3-4):355–372, 1996.

A Code

```
"""A Linear Regression baseline algorithm for predicting handwritten digits."""

import multiprocessing as mp
import pickle as pkl
from functools import partial
from random import gauss

import numpy as np
from sklearn.decomposition import PCA
from sklearn.linear_model import RidgeClassifier
from sklearn.metrics import (accuracy_score, classification_report,
                             confusion_matrix, precision_recall_fscore_support)
from sklearn.model_selection import ParameterGrid, RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler


def load_data(filename='mfeat-pix.txt'):
    """Load the data, make the labels."""
    data = np.loadtxt(filename)

    labels = np.zeros(data.shape[0], dtype=np.int)
    digitreps = int(data.shape[0] / 10)
    for digit in range(10):
        labels[digit * digitreps: (digit + 1) * digitreps] = digit

    return data, labels


def results(model, data, labels, show_list=[]):
    """Show and/or return the results of a model on some data given some labels.

    Print metrics by passing the names of the metrics via show_list like so:
    show_list=['prec', 'rec', 'F1', 'acc', 'MR', 'matrix', 'report']

    Always returns the following metrics:
    ['prec', 'rec', 'F1', 'acc', 'MR']

    """
    # Don't allow invalid metrics in show_list
    valid_metrics = ['prec', 'rec', 'F1', 'acc', 'MR', 'matrix', 'report']
    assert all(metric in valid_metrics for metric in show_list)
```

```
    # Gather the data
    pred = model.predict(data)  # This is the line that takes so long...
    precs, recs, F1s, _ = precision_recall_fscore_support(labels, pred)
    acc = accuracy_score(labels, pred)

    if show_list:
        # Helper function to print the mean and standard deviation of a list
        def show_mean_std(n, l):
            print(f'{n.title()}: mean={np.mean(l):.3f}, std={np.std(l):.3f}')

        # Possible metrics as partial functions
        show_dict = {
            'prec': partial(show_mean_std, 'precision', precs),
            'rec': partial(show_mean_std, 'recall', recs),
            'F1': partial(show_mean_std, 'f1-score', F1s),
            'acc': partial(print, f'Accuracy: {acc:.3f}'),
            'MR': partial(print, f'Misclassification Rate: {1 - acc:.3f}'),
            'matrix': partial(confusion_matrix, labels, pred),
            'report': partial(classification_report, labels, pred)
        }

        for metric in show_list:
            show_dict[metric]()

    return {
        'prec': [np.mean(precs), np.std(precs)],
        'rec': [np.mean(recs), np.std(recs)],
        'F1': [np.mean(F1s), np.std(F1s)],
        'acc': acc,
        'MR': 1 - acc,
    }


def clamp(value, minimum, maximum):
    """Clamp a value between a min and a max."""
    if minimum > value:
        value = minimum
    if maximum < value:
        value = maximum
    return value


def add_noise(x_train, y_train, spread=0, copies=1, keep_original=False):
    """Generate more data by adding noise.
```

```
    This function takes data = (x_train, y_train) and generates more data by
    making copies of the data with added Gaussian noise and the same label.
    """
    x_noise = []
    y_noise = []
    for digit, label in zip(x_train, y_train):
        # Keep the original if we want to
        if keep_original:
            x_noise.append(digit)
            y_noise.append(label)

        # Create copies with added noise
        for c in range(copies):
            # Every pixel is given a Gaussian noise component
            x_noise.append([clamp(pixel + gauss(0, spread), 0, 6)
                            for pixel in digit])
            y_noise.append(label)

    return x_noise, y_noise


def cross_val(pipeline, data, labels,
              n_splits=10, n_repeats=1,
              noise_spread=0, noise_copies=1):
    """Perform cross-validation while adding noise.

    This function brings the noise generation to the cross-validation to make
    sure that we are using noisy data for training only and not for testing.

    After each split, expand the training data by adding noise and then
    evaluate and record the performance.
    """
    cv = RepeatedStratifiedKFold(
        n_splits=n_splits,
        n_repeats=n_repeats,
        random_state=1,
    )

    # Let the CV split the data correctly, the correct number of times
    perf = list()
    for train_indices, test_indices in cv.split(data, labels):
        x_train, x_test = data[train_indices], data[test_indices]
        y_train, y_test = labels[train_indices], labels[test_indices]

        # Expand the data by adding noise to (x_train, y_train)
        x_train, y_train = add_noise(x_train, y_train,
```

```
                                        noise_spread, noise_copies)

            # Fit the model and collect the results (its performance)
            pipeline.fit(x_train, y_train)
            perf.append(results(pipeline, x_test, y_test))

    return perf


def execute_thread(pipeline, data, labels, params, classifier):
    """Delegate a set of params to multiprocessing."""
    if classifier == 'LR':
        pipeline.set_params(**{
            'pca__n_components': params['m'],
            'model__alpha': params['alpha'],
        })
    elif classifier == 'KNN':
        pipeline.set_params(**{
            'pca__n_components': params['m'],
            'model__n_neighbors': params['K'],
        })
    else:
        assert classifier in ['LR', 'KNN']

    print(params)

    perf = cross_val(pipeline, data, labels,
                     noise_spread=params['noise_spread'],
                     noise_copies=params['noise_copies'])

    return params, perf


def perform_experiment(exp, filename):
    """Perform the full experiment.

    Does a full parameter sweep for the experiment passed as a dictionary and
    saves the result to file. The experiment exp should define a pipeline, the
    parameters to sweep for, and give the name of the classifier.
    """
    # Load the data and labels
    data, labels = load_data()

    # Generate all the arguments for function calls in this parameter sweep
    args = [(exp['pipeline'], data, labels, params, exp['classifier'])
            for params in exp['parameters']]
```

```
    # Use multiprocessing to delegate functions calls across processors
    print(f'Delegating {len(args)} tasks to {mp.cpu_count()} cores')
    pool = mp.Pool(mp.cpu_count())
    results = pool.starmap(execute_thread, args)

    # Save results to file
    with open(filename, 'wb') as f:
        pkl.dump(results, f)

    return results


experiment_LR = {
    # Set the classifier name (must be one of {'LR', 'KNN'})
    'classifier': 'LR',

    # Define the pipeline
    'pipeline': Pipeline([
        ('normalize', StandardScaler()),
        ('pca', PCA()),
        ('model', RidgeClassifier()),
    ]),

    # Generate the parameter combinations to sweep for
    'parameters': list(ParameterGrid({
        'm': list(range(20, 51)),
        'alpha': [0],
        'noise_spread': np.arange(0, 3.25, 0.25),
        'noise_copies': [10],
    })),
}

experiment_KNN = {
    # Set the classifier name (must be one of {'LR', 'KNN'})
    'classifier': 'KNN',

    # Define the pipeline
    'pipeline': Pipeline([
        ('normalize', StandardScaler()),
        ('pca', PCA()),
        ('model', KNeighborsClassifier()),
    ]),

    # Generate the parameter combinations to sweep for
    'parameters': list(ParameterGrid({
```

```
        'm': list(range(20, 51)),
        'alpha': [0],
        'noise_spread': np.arange(0, 3.25, 0.25),
        'noise_copies': [10],
        'k': [3, 5, 7, 9],
    })),
}


if __name__ == '__main__':
    perform_experiment(experiment_LR, 'LR_results.pkl')
    perform_experiment(experiment_KNN, 'KNN_results.pkl')
```