Introduction to Machine Learning - Report 1

Alexandre Raybaut , Yassine El Graoui , Radu Tipurita

CS 233, EPFL

1. Introduction

This is a report stating the work we have done for the course project milestone 1. We have implemented three main methods: linear regression, logistic regression and KNN. Linear regression is used for regression, logistic regression for classification and KNN is used for both. We have also added two extra parameters in the main function:

1. Parameter `--k_fold` to run any method using cross_validation with $K = 5$. Please use the command :

```
python main.py --method method_name --task some_task --
  k_fold
```

   `method_name` can be replaced by `linear_regression`, `logistic_regression` and `knn`, so as to run the respective function.
2. Parameter `--with_kernel` to run linear regression with the kernel trick. Please use the command:

```
python main.py --method method_name --task some_task --
  with_kernel
```

2. Data Engineering

2.1. Data Pre-Processing

Firstly, we have randomly separated the training data into two groups: 80% of training data is dedicated to the train set and 20% of training data is dedicated to the validation set . Secondly, we pre-processed the data by normalising the data according to z-score normalisation, so that within features the data will be normally distributed.

The Z-Score formula is : $z^{(i)} = \frac{x^{(i)}-\mu}{\sigma}$

2.2. Data Transformation

We have added a bias to the data, hence the model can shift the prediction by some constant w_0. Additionally, we have added kernel data (all dot products with rbf kernel) to the pre-processing part, so as to use this kernel data if `--with_kernel` is passed to linear regression.

3. Models

3.1. Linear Regression

We have added cross validation for linear regression to select the best hyper parameter λ from a list of candidates chosen exponentially: powers of 10 going from 10^{-9} to 100.

Cross validation gives us the following plot:

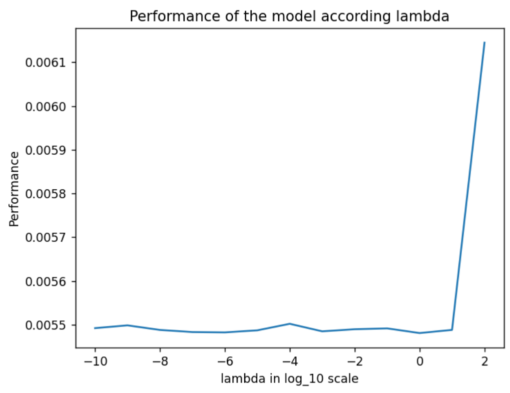

Figure 1. On the plot, we can see that the optimal λ in our list is 1. When also considering $\lambda = 0$, we notice that the result does not change, i.e $\lambda = 1$ is still optimal with a MSE of 0.00548059

3.2. Logistic regression

We have added cross validation to select the two best combinations of learning rate `lr` and max iterations `max_iters`. Instead of cross validating the two separately, we cross validate them simultaneously, because some `max_iters` are too low for some learning rates η, such combinations yielding poor results. After selecting the optimal combination `lr` and `max_iters`, we do another round of cross validation to select the best possible standard deviation (hyper-parameter σ) for the normal initialisation of the weights.

Cross validation gives us the following plot:

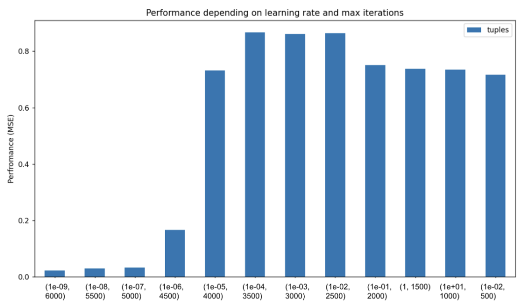

Figure 2. The optimal combination is : max iters : 3500; Learning rate : 0.0001. The resulting maximum F1-score is: 0.8662849263534218

3.3. KNN

F1 Score is more suitable for imbalanced data than accuracy, as it is the harmonic mean between precision and recall: it takes into account the distribution of the data.

$$F_1 = \frac{2}{precision^{-1} + recall^{-1}}$$

By inspecting the data, we noticed that there can be a gap of 100 dogs between different classes. For example, there are 215 dogs of breed 7 but only 129 of breed 18. Thus, the F1-score might be a better metric than the accuracy that does not take care of this data imbalance. We have added cross validation to select the best combination of the following hyper-parameters : distance metric, weighting function and k. We have the following choices :

- For k, we have the list of positive integers going from 1 to 26, which is the number of closest neighbours taken into account in a prediction.
- For the distance functions, we have three options which are euclidean distance, minkowski distance also know as L_p (with $p = 4$ to keep a relatively simple model, with a convex distance metric) and kernelised distance based on Gaussian radial basis function:

  Euclidean distance: $d(x, y) = \|x - y\|_2^2$
  Minkowski distance:

$$d(x, y) = \|x - y\|_p^p = (\sum_{i=1}^{D} |x_i - y_i|^p)^{1/p}$$

  Gaussian kernel distance: $d(x, y) = K(x, y) = \exp(-\frac{\|x-y\|_2^2}{2\sigma^2})$
- For the weighting function, we have two options, either the identity function (hence all weights are one, no weighting) or the decaying weights function $e^{-distances}$.

To choose the best hyper-parameter, we decided to take the highest F1-score if the task selected was classification, or the lowest MSE if the task selected was regression. For that, we create a 3D tensor with the different dimensions being the distance functions, the weighting functions and the chosen K.

This gives us these three options, for classification :

- $k = 16$, distance Function = Minkowski, Weighting function = Identity
- $k = 14$, distance Function = Euclidean distance, Weighting function = Identity
- $k = 17$, distance Function = Minkowski, Weighting function = Identity

The best combination ($k = 16$, Minkowski, Identity) has a result of :

```
            accuracy = 99.957% - F1-score = 0.999521
            Test set:  accuracy = 83.333% - F1-score = 0.833503
```

The command to recreate our results:

```
            python main.py --task breed_identifying --method knn
      --k_fold
```

And, these three options for regression :

- $k = 9$, distance Function = Gaussian RBF, Weighting function = Decaying weights
- $k = 25$, distance Function = Gaussian RBF, Weighting function = Identity
- $k = 8$, distance Function = Gaussian RBF, Weighting function = Decaying weights

The best combination ($k = 9$, Gaussian RBF, Decaying weights) has a result of :

```
MSE = 0.00768491, Train loss = 0.008% - Test loss 0.008.
```

The command to recreate our results:

```
   python main.py --method knn --task center_locating --k_fold
```

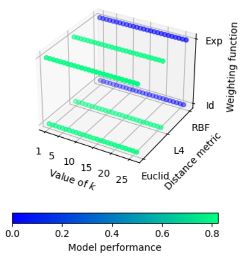

Figure 3. Measurement of the F1-score depending on the distance function, k and the weighting function

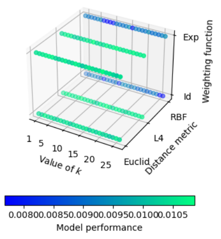

Figure 4. Measurement of the MSE depending on the distance function, k and the weighting function

4. Discussion

First of all, when implementing the basic functionalities, two interesting cases appeared:

- In the linear regression class, we had to distinguish between the case $\lambda = 0$ and the case $\lambda \neq 0$. Indeed, computing an inverse is less numerically stable than solving a linear system (with a square matrix). It would also be less expensive, as we don't have to first calculate the inverse $O(n^3)$ and then a matrix vector product $O(n^2)$. Thus we have more precise results when using the function:

```
np.linalg.solve(X.T@X + self.lmda * np.eye(D), X.T @
      training_labels)
```

  The reason why solving the linear system is always possible is the following: the matrix $\Phi^T\Phi$ is positive semi definite, i.e for all vectors v: $v^T\Phi^T\Phi v = (\Phi v)^T\Phi v = \|\Phi v\|_2^2 \geq 0$. Hence we know that all of its eigenvalues are greater than 0: $v^T\Phi^T\Phi v \geq 0 \Rightarrow v^T\Phi^T\Phi v = v^T\lambda v = \lambda\|\Phi v\|_2^2 \geq 0 \Rightarrow \lambda \geq 0$. Hence using the ortho-diagonalising the matrix (possible since it's symmetric) $\Phi^T\Phi = PDP^T \Rightarrow \Phi^T\Phi + \lambda I = PDP^T + P\lambda IP^T \Rightarrow \Phi^T\Phi = P(D + \lambda)P^T$. Since $\lambda > 0$ we immediately know that all eigenvalues of $\Phi^T\Phi + \lambda I$ are stricly positive, i.e $\forall i : \lambda_i > 0$ and thus the determinant is non zero and the matrix is invertible (a solution to the linear system always exists). However, when $\lambda = 0$, we cannot apply the latter formula, because the matrix is positive semi definite, i.e $\forall i : \lambda_i \geq 0$, it may have a zero determinant and not be invertible. So, we had to do a case disjunction to keep an consistent model. When, $\lambda \neq 0$ we use `np.linalg.solve`, otherwise we use the pseudoinverse `np.pinv`.
- In the logistic regression class, we had an overflow when computing the f_softmax function, since we exponentiate a matrix. So, we can subtract the maximum of each line and then exponentiate to avoid overflow. Doing that, we only have negative numbers or zeros which allow us to work with small values when exponentiated. This causes another problem : there can be a line that underflows and that sums to zero (since we have negative numbers that are exponentiated). A solution can be the following : If the denominator is zero (so it underflowed), then necessarily the numerator must be zero. Indeed, it is by definition a probability, hence between 0 and 1. Thus, in the case of a sum equal to zero, the result should be zero. If the sum is not zero, then no underflow occurred. We can compute the usual formula. The first part of this point was answered through this website : https://blester125.com/blog/softmax.html#:~:text=If%20one%20of%20your%20elements,subsequent%20sum%20will%20never%20overflow
- We have decided to implement the kernel trick for two of our models, knn and linear regression: the only kernel we tried out was gaussian RBF (as stated above). We have a general radial basis function in `main.py` that can compute all radial dot products between vectors in two different matrices. This allows us to kernelised training set and kernelised test set. That being said, the kernel trick did not lead to better results in `linear_regression`, as the model was able to generalise in the input space, so a feature expansion was not necessary:

```
python main.py --method linear_regression --lmda 1 --
      with_kernel
```

```
------------ Kernel ----------------
mse_loss_with_kernel_trainings: 0.005405269595694235
mse_loss_with_kernel_testing: 0.006538908703367491
------------ Kernel ----------------
Train loss = 0.005% - Test loss = 0.005
```

  Gaussian kernel distance leaded to worse results than other distance metrics in knn for classification as illustrated in **Figure 3**: average F1-score much lower than for other distance metrics. However, we saw an improvement due to a gaussian kernel in knn for regression as illustrated **Figure 4**: the average MSE for a gaussian kernel tends to be on average lower than for other distance metrics.

Furthermore, we decided that it would be much better to cross-validate our hyper parameters rather than just blindly trying by hand to find an heuristically optimal solution. Hence we implemented a cross validation, with K-fold, where $K = 5$, to find the best possible hyper paremeter value combinations. For that, we created a 3D tensor for knn with 3 axes, a 1D vector for both logistic regression and linear. Each entries of these nd-arrays (ND tensors), where each dimension corresponded to a hyper-parameter, were filled with the relevant performance metric (if we were doing a regression task we used MSE and if we were doing classification we used F1-score). Finally, we unravel all the tensor into a single 1D array from which we take the wanted best scores of the metric. The only special case is in logistic regression, where max_iters and lr are interdependent, one being expressed as a function of the other. Indeed, low learning rates coupled with low `max_iters` would perform poorly, as gradient descent would not be able to converge and high learning rates coupled with high `max_iters` would give insignificant results, as gradient descent would just diverge . This is why we formed couples as follows:

```
index_list = np.arange(12)
lr_list = np.float_power(10,  (index_list - 9))
max_iters_list = 500 * (index_list[::-1] + 1)
```

Also, we could optimise the p parameter in the Minkowski distance metric (L_p norm, though it would likely overfit the training data, by being too specific on slight differences in input samples specific to the p-dimensional space, the distance was computed in). Another drawback is that we would have an even slower model. We concluded that introducing this additional hyper-parameter was not necessary, since the performance was already obtained a good enough with MSE/F1-score.
Finally after getting our results, we tried to plot them to visualise the difference between the models (and the different hyperparameter combinations).