Machine Learning
Home Assignment 4

Carlo Rosso rkm957

Contents

1 Sleep well

1.1 Data understanding and preprocessing

Follows the code I used to compute the frequency of each sleep stage in the dataset and the table with the results:

```
import pandas as pd
train_data = pd.read_csv("path/to/dataset/X_train.csv", header=None)
print(train_label.value_counts() / len(train_label))
```

Note that the path to the dataset is a placeholder and should be replaced with the actual path to the dataset, I uploaded the dataset to google drive, that is why I replaced it here with a placeholder. The function `value_counts()` returns the number of occurrences of each unique value in the dataset.

Sleep Stages	Frequency
0	0.5209
1	0.0955
2	0.2527
3	0.0469
4	0.0839

Table 1: Frequency of sleep stages in the dataset

1.2 Classification

First of all I defined the following function:

```
def zol(model, prefix = ""):
  print(prefix, zero_one_loss(train_label, model.predict(train_data)))
  print(prefix, zero_one_loss(test_label, model.predict(test_data)))
```

which uses the `zero_one_loss` function from the `sklearn.metrics` module to compute the zero-one loss of the model on the training and test data, because this is asked in each of the following tasks.

1.2.1 Logistic Regression

```
from sklearn.linear_model import LogisticRegression as lr

logistic = lr(solver = "lbfgs", max_iter = 500)
logistic.fit(train_data, train_label)
zol(logistic)
```

Then I used the `LogisticRegression` class from the `sklearn.linear_model` module to train a logistic regression model. I changed the solver to the one used by the professor, in his notebook[1].

I augment the number of iterations to 500, because the default value of 100 was not enough for the model to converge and I keep the default penalty of the weight term, which is L2.

Follows the training and test error of the model:

Dataset name	Loss
Training	0.1496
Test	0.0993

Table 2: Zero-one loss of the logistic regression model

1.2.2 Random forest

```
from  sklearn.ensemble import RandomForestClassifier as rf
t = rf(n_estimators = nTrees, oob_score=True)
t.fit(train_data, train_label)
zol(t)
print(t.oob_score_)
```

I use the `RandomForestClassifier` class from the `sklearn.ensemble` module to train the random forest models.

Note that the `nTrees` variable was given in the assignment, while I choose to keep the default values for the other parameters. In particular, I choose to use `gini` to measure the impurity for the classification trees (we discussed it in the lecture) and I keep the `m`, which is the maximum number of features to consider when looking for the best split, to the default value of `sqrt`, as the professor suggests.

Follows the training and test error of the models and the out-of-bag error:

N. of Trees	Training One Zero Loss	Test One Zero Loss	Out-of-bag error
50	0.0001186	0.1145	0.1527
100	0.0000	0.1111	0.1495
200	0.0000	0.1118	0.1481

Table 3: Zero-one loss of the random forest model

1.2.3 Nearest neighbour

```
param_grid = {'n_neighbors': [i**2 for i in range(1, int(len(train_label)
** (1/4)))]}
grid_search = GridSearchCV(knn(), param_grid, cv=5, scoring="accuracy")
grid_search.fit(train_data, train_label)
print(grid_search.best_params_['n_neighbors'])
zol(grid_search.best_estimator_)
```

I used the `GridSearchCV` class from the `sklearn.model_selection` module to determine the number of neighbors. In particular, `CV` of `GridSearchCV` stands for cross-validation: you provide the name of the hyperparameter you want to tune and the values you want to try, and the class will try all the possible combinations of the hyperparameters. In this case, we are only interested in the best model, so we can access it through the `best_estimator_` attribute. Note that `param_grid` has length equal to 12, indeed it contains all the squares of the numbers from 1 to 12.

Dataset Name	Zero One Loss
Training set	0.1497
Test set	0.0957

Table 4: Zero-one loss of the nearest neighbour model

2 Invariance and normalization

2.1 Nearest neighbour

The Nearest Neighbour algorithm is affected by standardization, which is the process of transforming a dataset so that its mean is 0 and its variance (or standard deviation) is 1. As an example, I train the same model as in the previous task, but with the standardized dataset and as a result I obtain different training and test losses:

Model name	Training Zero One Loss	Testing Zero One Loss
Given Training set	0.1580	0.0984
Standardized Training set	0.1486	0.0970

Table 5: Zero-one loss of nearest neighbour models tested on standardized data

Indeed changing the scale of the feature in the dataset changes the norm, and so the distance between the points, which is the metric used by the Nearest Neighbour algorithm to classify the points.

2.2 Logistic regression

The logistic regression model is affected by the standardization of the dataset, meaning it is not invariant to linear transformations on the data. On the other hand, retraining the model on a standardized dataset should not change its overall performance. This is because the logistic regression works by applying a weighted sum of the input, therefore the model's weights adjust to the new scale of the features. Meanwhile centering the dataset (i.e. subtracting the mean) does not affect the model, because the bias term compasates for the shift.

In my experiments, I trained logistic regression models on both standardized and non-standardized datasets, applying the default L2 regularization and also training without any penalty terms. The results varied in each scenario. Notably, there was no significant difference between the models with and without L2 regularization, but a clear difference was observed between models trained on standardized versus non-standardized data. This difference cannot be attributed to the scaling of the loss function, since the zero-one loss, which depends solely on the model's predictions, remains unaffected by feature scaling.

Follows the results of the experiments:

Model name	Training Zero One Loss	Testing Zero One Loss
Given Training set + L2	0.1496	0.0993
Standardized Training set + L2	0.1497	0.1024
Given Training set	0.1498	0.0995
Standardized Training set	0.1497	0.1024

Table 6: Zero-one loss of logistic regression models

Note that I the zero one loss is computed in the same dataset as the model is trained in. What I mean is that I am not verifying the invariance of the model. In the notebook, I also test the invariance, and of course, I find out that the models are not invariant to the standardization of the dataset.

2.3 Random forest

The Random Forest algorithm is affected by the standardization of the dataset in the meaning it is not invariant to linear transformations on the dataset. On the other hand, retraining the model on a standardized dataset should not change its overall performance. This is because the Random Forest algorithm works by averaging the predictions of multiple decision trees. And a decision tree is built by splitting the feature space in a way that

minimizes the impurity of the nodes. Therefore, the model's weights adjust to the new scale of the features.
I also experimented with this model and I also obtained different results, I think it is due to the fact that I did not set the `random_state` parameter.

3 Differentiable programming

3.1 Migration to Pytorch

```
# Define starting point in the upper right corner of plot
xi = torch.tensor([0.9*r], requires_grad=True)
yi = torch.tensor([0.8*r], requires_grad=True)
p_x = [xi.item()]  # list of x-values
p_y = [yi.item()]  # list of y-values
optimizer = torch.optim.SGD([xi, yi], lr=eta)

# Do steepest descent optimization:
for i in range(n_iter):
  optimizer.zero_grad()
  loss = f(xi, yi)
  loss.backward()
  optimizer.step()
  p_x.append(xi.item())  # store x-coordinate
  p_y.append(yi.item())  # store y-coordinate
```

As suggested in the assignment, I converted `xi` and `yi` to PyTorch tensors, then I initialized the stochastic gradient descent optimizer with the learning rate `eta` and the tensors `xi` and `yi`. Afterwards, following the provided documentation, I implemented the actual training loop:

1. reset the gradients;
2. define the function `f` to be optimized;
3. backpropagate the losses backwards to each parameter;
4. update the parameters with the optimizer;
5. store the updated parameters in the lists `p_x` and `p_y`, to have a record of

the "path" the optimizer took.

I checked the produced plot and the arrays `p_x` and `p_y` prior and after the migration and they are identical, so I am confident that the migration was successful.

Bibliography

[1] C. Igel, “Decision Functions of Multi-class Logistic Regression.” [Online]. Available: https://github.com/christian-igel/ML/blob/main/notebooks/MLA/Multi-class%20logistic%20regression.ipynb