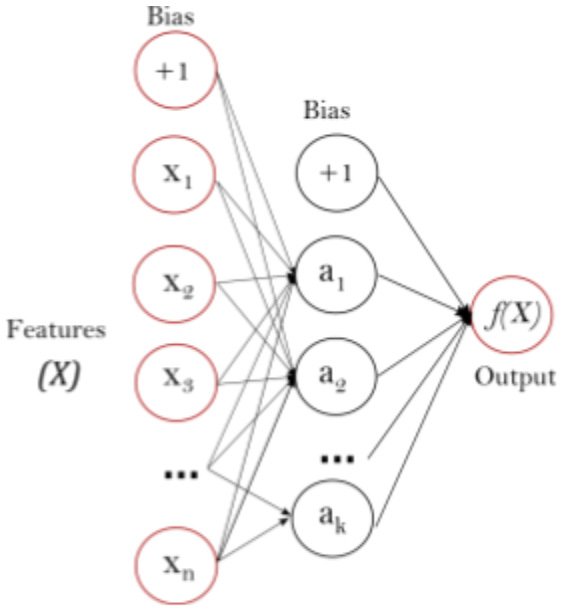
Comparison of different meta-heuristic methods to optimize coefficients of a neural network to solve the benchmark classification problem

Maryam Alipourhajiagha
Artificial Intelligence Course Assignment
Amirkabir University of Technology

Introduction

It is not debatable that Classification plays a vital role in Data Science and Machine Learning problems. Training models to classify and categorize diverse data types are popular among data scientists. Classification is one of the many applications in which Artificial Neural Networks, or ANNs for short, are utilized nowadays. This report will discuss three solvers of one of the easiest-to-implement Neural Networks for classification from Scikit-Learn called the Multi-layer Perceptron classifier (MLPClassifier).

MLP is a supervised learning algorithm composed of more than one perceptron. MLPs include an input layer to receive the signal and an output layer that gives the target variable, and in between those two, there are other layers that are the true computational engine of the MLP.



one hidden layer MLP with scalar output

MLPClassifier is a class that implements an MLP algorithm that trains using different optimization algorithms to update model parameters. We cover Adam, L-BFGS, and Stochastic Gradient Descent (SGD) optimizers.

Drug Classification Problem

Consider yourself a medical researcher gathering information for a study. You have collected data on a group of patients who are all dealing with the same illness. During treatment, each patient responds positively to one of the five medications, A, X, C, B, and Y, and is treated.
To determine the best medication for a new patient with the same condition, we wish to develop a model. The aim of this data set is to locate a drug that each patient may be treated with.

Reading Data

First of all, I read the data through of pandas package. The features of this data set include the age, sex, blood pressure, and cholesterol of the patients.

[1]:
```
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPClassifier
```

[2]:
```
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
my_data = pd.read_csv("drug200.csv", delimiter=",")
my_data.head()
```

[3]:

	Age	Sex	BP	Cholesterol	Na_to_K	Drug
0	23	F	HIGH	HIGH	25.355	drugY
1	47	M	LOW	HIGH	13.093	drugC
2	47	M	LOW	HIGH	10.114	drugC
3	28	F	NORMAL	HIGH	7.798	drugX
4	61	F	LOW	HIGH	18.043	drugY

Data Pre-Processing

Now, with one hot encoding method, I convert the sex and bp column into a new categorical column and assign a binary value of 1 or 0 to those columns. After that, I drop the target column from the primary data frame and keep it in the y vector.

[4]:
```
X = my_data[['Age', 'Sex', 'BP', 'Cholesterol', 'Na_to_K']].values
X[0:5]
```

[4]:
```
array([[23, 'F', 'HIGH', 'HIGH', 25.355],
       [47, 'M', 'LOW', 'HIGH', 13.093],
       [47, 'M', 'LOW', 'HIGH', 10.114],
       [28, 'F', 'NORMAL', 'HIGH', 7.798],
       [61, 'F', 'LOW', 'HIGH', 18.043]], dtype=object)
```

[5]:
```
from sklearn import preprocessing
le_sex = preprocessing.LabelEncoder()
le_sex.fit(['F','M'])
X[:,1] = le_sex.transform(X[:,1])


le_BP = preprocessing.LabelEncoder()
le_BP.fit([ 'LOW', 'NORMAL', 'HIGH'])
X[:,2] = le_BP.transform(X[:,2])


le_Chol = preprocessing.LabelEncoder()
le_Chol.fit([ 'NORMAL', 'HIGH'])
X[:,3] = le_Chol.transform(X[:,3])

X[0:5]
```

[5]:
```
array([[23, 0, 0, 0, 25.355],
       [47, 1, 1, 0, 13.093],
       [47, 1, 1, 0, 10.114],
       [28, 0, 2, 0, 7.798],
       [61, 0, 1, 0, 18.043]], dtype=object)
```

[6]:
```
y = my_data["Drug"]
y[0:5]
```

[6]:
```
0    drugY
1    drugC
2    drugC
3    drugX
4    drugY
Name: Drug, dtype: object
```

Finally, we use the Multi-layer perceptron model from the scikit-learn module. The "solver" argument determines the optimizer type.

LBFGS Solver

It is an optimizer from the Quasi-Newton family and stands for Limited-memory Broyden–Fletcher–Goldfarb–Shanno. LBFGS calculates an approximation to the Hessian matrix (Second-Derivate matrix) based on the gradient. It is efficient in memory use but is slow in training big sets of data.

Training

[19]:
```
MLP = MLPClassifier(solver='lbfgs')
MLP # it shows the default parameters
```

[19]:
```
MLPClassifier(solver='lbfgs')
```

[20]:
```
MLP.fit(X_trainset,y_trainset)
```

[20]:
```
MLPClassifier(solver='lbfgs')
```

Predicting

[21]:
```
predMlp = MLP.predict(X_testset)
```

[22]:
```
print (predMlp [0:5])
print (y_testset [0:5])
```

```
['drugY' 'drugX' 'drugX' 'drugX' 'drugX']
40     drugY
51     drugX
139    drugX
197    drugX
170    drugX
Name: Drug, dtype: object
```

Evaluation

[23]:
```
from sklearn import metrics
import matplotlib.pyplot as plt
print("Accuracy: ", metrics.accuracy_score(y_testset, predMlp))
print(metrics.classification_report(y_testset, predMlp))
```

```
Accuracy:  0.8833333333333333
              precision    recall  f1-score   support

       drugA       1.00      1.00      1.00         7
       drugB       0.62      1.00      0.77         5
       drugC       1.00      0.80      0.89         5
       drugX       0.95      0.90      0.93        21
       drugY       0.86      0.82      0.84        22

    accuracy                           0.88        60
   macro avg       0.89      0.90      0.88        60
weighted avg       0.90      0.88      0.89        60
```

SGD Solver

Stochastic Gradient Descent, known as SGD, is an iterative technique for optimizing an objective function. It can be viewed as a stochastic approximation of gradient descent optimization since it substitutes the actual gradient, which is derived from the complete data set, with an estimated gradient, which is generated from a randomly chosen portion of the data. SGD solved the Gradient Descent problem by parameter updating from a single record. SGD is slow to converge because it necessitates forward and backward propagation for every record. The route to the global minima also becomes very unpredictable.

Training

```
[9]: MLP = MLPClassifier(solver='sgd')
     MLP # it shows the default parameters
```

```
[9]: MLPClassifier(solver='sgd')
```

```
[10]: MLP.fit(X_trainset,y_trainset)
```

```
[10]: MLPClassifier(solver='sgd')
```

Predicting

```
[11]: predMlp = MLP.predict(X_testset)
```

```
[12]: print (predMlp [0:5])
      print (y_testset [0:5])
```

```
['drugX' 'drugX' 'drugX' 'drugX' 'drugY']
40     drugY
51     drugX
139    drugX
197    drugX
170    drugX
Name: Drug, dtype: object
```

Evaluation

```
[13]: from sklearn import metrics
      import matplotlib.pyplot as plt
      print("Accuracy: ", metrics.accuracy_score(y_testset, predMlp))
      print(metrics.classification_report(y_testset, predMlp))
```

```
Accuracy:  0.6333333333333333
              precision    recall  f1-score   support

       drugA       0.00      0.00      0.00         7
       drugB       1.00      0.60      0.75         5
       drugC       0.00      0.00      0.00         5
       drugX       0.73      0.76      0.74        21
       drugY       0.54      0.86      0.67        22

    accuracy                           0.63        60
   macro avg       0.45      0.45      0.43        60
weighted avg       0.54      0.63      0.57        60
```

Adam Solver

Adam is a technique for optimizing stochastic objective functions using first-order gradients that are based on adaptive estimations of lower-order moments. The method is suitable for situations with enormous amounts of data and/or parameters since it is computationally efficient, requires little memory, and is invariant to the diagonal rescaling of the gradients. The approach is also suitable for non-stationary goals and issues with extremely noisy and/or sparse gradients. The hyper-parameters may usually be tuned to a reasonable degree and have intuitive interpretations.

Training

[14]:
```
MLP = MLPClassifier(solver='adam')
MLP # it shows the default parameters
```

[14]:
```
MLPClassifier()
```

[15]:
```
MLP.fit(X_trainset,y_trainset)
```

[15]:
```
MLPClassifier()
```

Predicting

[16]:
```
predMlp = MLP.predict(X_testset)
```

[17]:
```
print (predMlp [0:5])
print (y_testset [0:5])
```

```
['drugY' 'drugX' 'drugX' 'drugX' 'drugY']
40     drugY
51     drugX
139    drugX
197    drugX
170    drugX
Name: Drug, dtype: object
```

Evaluation

[18]:
```
from sklearn import metrics
import matplotlib.pyplot as plt
print("Accuracy: ", metrics.accuracy_score(y_testset, predMlp))
print(metrics.classification_report(y_testset, predMlp))
```

```
Accuracy:  0.6666666666666666
              precision    recall  f1-score   support

       drugA       0.00      0.00      0.00         7
       drugB       0.67      0.80      0.73         5
       drugC       0.00      0.00      0.00         5
       drugX       0.82      0.86      0.84        21
       drugY       0.58      0.82      0.68        22

    accuracy                           0.67        60
   macro avg       0.41      0.50      0.45        60
weighted avg       0.55      0.67      0.60        60
```

Conclusion

When True Positives and True Negatives are more important than False Positives and False Negatives, accuracy should be employed instead of F1-score. When the class distribution is similar, accuracy can be employed; however, when there are imbalanced classes, as in the example above, F1-score is a preferable metric. According to the evaluation result, the F1-score of the model which uses the LBFGS optimizer is 0.88. Compared to the Adam and SGD solver, LBFGS's F1-score is 0.2 higher.