CSE5001 AAI: Assignment3 Report

Name: 吉辰卿 SID: 12332152

Part 1: Data Preprocessing

When we get the training data set, we find that the columns **have both numerical and textual features**. Therefore, our first step is to **encode these features properly so that they can be input to a machine learning model**. For **most of the text features**, we can adopt the `LabelEncoder` in `sklearn.preprocessing`, as shown below:

```python
# 数据预处理--编码
# relabel name information
name_column = data['Name']
label_encoder = LabelEncoder()
encoded_names = label_encoder.fit_transform(name_column)
# 将编码后的结果添加到data中
data['Name'] = encoded_names

# relabel Occupation information
occupation_column = data['Occupation']
label_encoder = LabelEncoder()
encoded_occupations = label_encoder.fit_transform(occupation_column)
# 将编码后的结果添加到data中
data['Occupation'] = encoded_occupations

# relabel Credit_Mix information
credit_mix_column = data['Credit_Mix']
label_encoder = LabelEncoder()
encoded_credit_mix = label_encoder.fit_transform(credit_mix_column)
# 将编码后的结果添加到data中
data['Credit_Mix'] = encoded_credit_mix

# relabel Payment_Behaviour information
payment_behaviour_column = data['Payment_Behaviour']
label_encoder = LabelEncoder()
encoded_payment_behaviour =
label_encoder.fit_transform(payment_behaviour_column)
# 将编码后的结果添加到data中
data['Payment_Behaviour'] = encoded_payment_behaviour

# relabel Payment_of_Min_Amount information
payment_of_min_amount_column = data['Payment_of_Min_Amount']
label_encoder = LabelEncoder()
encoded_payment_of_min_amount =
label_encoder.fit_transform(payment_of_min_amount_column)
# 将编码后的结果添加到data中
data['Payment_of_Min_Amount'] = encoded_payment_of_min_amount
```

In the above code, we encode the feature `Name` , `Occupation` , `Credit_Mix` , `Payment_Behaviour` and `Payment_of_Min_Amount` using the labelencoder. **LabelEncoder uses different numeric values to encode all possible text values in a feature. For the same text value in a feature, the encoded value is the same.**

However, there is a column of text that cannot use the `LabelEncoder` for encoding, this column is `Type_of_Loan` column. We can see that in this column, **there are multiple text values in each row, separated by commas.** Since there are multiple properties for the eigenvalues of each row in this column, we can use the `MultiLabelBinarizer` in `sklearn.preprocessing` to encode the different texts in this column, as shown below:

```
# relabel Type_of_Loan information
type_of_Loan_column = data['Type_of_Loan']
labels_column = type_of_Loan_column.str.split(',')
label_encoder = MultiLabelBinarizer()
encoded_type_of_Loan = label_encoder.fit_transform(labels_column)
# 创建包含编码后结果的新DataFrame
encoded_df = pd.DataFrame(encoded_type_of_Loan, columns =
label_encoder.classes_)
# 将编码后的结果添加到原始DataFrame中
type_of_loan_index = data.columns.get_loc('Type_of_Loan')
data = pd.concat([data.iloc[:, :type_of_loan_index + 1], encoded_df,
data.iloc[:, type_of_loan_index + 1:]], axis=1)
# 删除原始包含多个标签的列
data.drop(columns=['Type_of_Loan'], inplace=True)
```

As we can see, when encoding the `Type_of_Loan` column, we first need to separate all the text values in each row of the column, and then **use all of those text values as the new feature columns. If the original `Type_of_Loan` column line has this text value feature, the corresponding feature column change line value is 1, otherwise 0.** Finally, we delete the original `Type_of_Loan` feature column and append these new feature columns encoded with binary values to the original feature column.

At the same time, we observe that the label column `Credit_Score` of this dataset has three different text values (`Good` , `Standard` and `Poor`). So we encode these three different text values separately with three numbers (e.g. 0, 1, 2), as shown below:

```
# relabel Credit_Score(label column) information
data.loc[data.Credit_Score == 'Good', 'Credit_Score'] = 0
data.loc[data.Credit_Score == 'Standard', 'Credit_Score'] = 1
data.loc[data.Credit_Score == 'Poor', 'Credit_Score'] = 2
```

Moreover, we find that in this dataset, the values of the columns `Customer_ID` , and `SSN` have duplicated information compared with the `Name` column we encoded earlier. Therefore, in order to avoid unnecessary duplication of coding, we will delete these columns. Also, the value of the `ID` column and its `previous column (the first column)` has no relationship to the label column `Credit_Score` . So we delete these two columns, as shown below:

```python
# 扔掉第一列
data = data.iloc[:, 1:]
print(data.shape)
##确定要扔掉的列
column_to_drop = ['ID','Customer_ID','SSN']
for col in column_to_drop:
    data = data.drop(columns= col)
```

Note that in this section, **the test set file is preprocessed using exactly the same coding method as the training set file.** (The only difference is that the label column in test set file does not have a corresponding value, so we do not need to code the label column in test set file), the related codes are shown below:

```python
## read and clean the testing dataset
data = pd.read_csv('test_set.csv')
# glimpse at the train data
data.head()
data.info()

# 数据清洗
# relabel name information
name_column = data['Name']
label_encoder = LabelEncoder()
encoded_names = label_encoder.fit_transform(name_column)
# 将编码后的结果添加到data中
data['Name'] = encoded_names

# relabel Occupation information
occupation_column = data['Occupation']
label_encoder = LabelEncoder()
encoded_occupations = label_encoder.fit_transform(occupation_column)
# 将编码后的结果添加到data中
data['Occupation'] = encoded_occupations

# relabel Type_of_Loan information
type_of_Loan_column = data['Type_of_Loan']
labels_column = type_of_Loan_column.str.split(',')
label_encoder = MultiLabelBinarizer()
encoded_type_of_Loan = label_encoder.fit_transform(labels_column)
# 创建包含编码后结果的新DataFrame
encoded_df = pd.DataFrame(encoded_type_of_Loan, columns =
label_encoder.classes_)
# 将编码后的结果添加到原始DataFrame中
type_of_loan_index = data.columns.get_loc('Type_of_Loan')
data = pd.concat([data.iloc[:, :type_of_loan_index + 1], encoded_df,
data.iloc[:, type_of_loan_index + 1:]], axis=1)
# 删除原始包含多个标签的列
data.drop(columns=['Type_of_Loan'], inplace=True)

# relabel Credit_Mix information
credit_mix_column = data['Credit_Mix']
label_encoder = LabelEncoder()
encoded_credit_mix = label_encoder.fit_transform(credit_mix_column)
# 将编码后的结果添加到data中
```

```python
data['Credit_Mix'] = encoded_credit_mix

# relabel Payment_Behaviour information
payment_behaviour_column = data['Payment_Behaviour']
label_encoder = LabelEncoder()
encoded_payment_behaviour = 
label_encoder.fit_transform(payment_behaviour_column)
# 将编码后的结果添加到data中
data['Payment_Behaviour'] = encoded_payment_behaviour

# relabel Payment_of_Min_Amount information
payment_of_min_amount_column = data['Payment_of_Min_Amount']
label_encoder = LabelEncoder()
encoded_payment_of_min_amount = 
label_encoder.fit_transform(payment_of_min_amount_column)
# 将编码后的结果添加到data中
data['Payment_of_Min_Amount'] = encoded_payment_of_min_amount

# relabel Credit_Score(label column) information
data.loc[data.Credit_Score == 'Good', 'Credit_Score'] = 0
data.loc[data.Credit_Score == 'Standard', 'Credit_Score'] = 1
data.loc[data.Credit_Score == 'Poor', 'Credit_Score'] = 2
print(data.shape)

## 重新输出data到csv检查一下数据是否成功清洗了
# 扔掉第一列
data = data.iloc[:, 1:]
print(data.shape)

##确定要扔掉的列
column_to_drop = ['ID','Customer_ID','SSN']
for col in column_to_drop:
    data = data.drop(columns= col)
print(data.shape)

data = data.to_numpy(dtype=np.float32)
#对挑选的列进行归一化
for ii in range(data.shape[1]-1):
    meanVal=np.mean(data[:,ii])
    stdVal=np.std(data[:,ii])
    data[:,ii]=(data[:,ii]-meanVal)/stdVal
```

Finally, we normalize each column in the encoded dataset as follows:

```python
data = data.to_numpy(dtype=np.float32)
#对挑选的列进行归一化
for ii in range(data.shape[1]-1):
    meanVal=np.mean(data[:,ii])
    stdVal=np.std(data[:,ii])
    data[:,ii]=(data[:,ii]-meanVal)/stdVal
```

Part 2: Preparation of the Datasets

First, we divide the encoded training set file into training sets and validation sets according to 8:2 by using the `train_test_split` in `sklearn.model_selection`, as follows:

```python
train, val = train_test_split(data, test_size =0.2, random_state=27893,stratify=data[:,-1])
```

Since the test sets does not need to be divided, we can directly use the encoded test set file as the test sets.

```python
test = data
```

Later, since we implemented the MLP model with `Pytorch`. We have to design the data sets that conforms to `DataLoader` in `torch.utils.data`, so we also need to define a data set class of our own, like this:

```python
class Diabetes_dataset(torch.utils.data.Dataset):
  def __init__(self, setname):
    self.setname=setname
    if setname=='train':
      X=train
    elif setname=='val':
      X=val
    elif setname=='test':
      X=test
    self.X = X

  def __len__(self):
    return len(self.X)

  def __getitem__(self, index):
    if (self.setname=='train') or (self.setname=='val'):
      data = self.X[index,0:-1]
      data = torch.tensor(data, dtype=torch.float32)
      label = self.X[index,-1]
      label = torch.tensor(label, dtype=torch.int64)
      data_pair = {'X': data, 'y': label}
    elif self.setname=='test':
      data = self.X[index,0:-1]
      data = torch.tensor(data, dtype=torch.float32)
      data_pair = {'X': data}
    return data_pair
```

Notice that the above dataset class inherits the `torch.utils.data.Dataset` parent class, which is a dataset class specified in Pytorch, in order to facilitate the training process by loading it with `DataLoader` class.

Part 3: Models Selection

In this section, we select `5 machine learning method`s and `1 neural network method`. The introduction of each model is as follows:

- **Decision Tree**

  We use the `tree.DecisionTreeClassifier` module in `sklearn` to create the decision tree model directly.

  ```
  ###使用决策树模型评估
  from sklearn import tree
  decision_tree = tree.DecisionTreeClassifier(criterion='entropy')      #实例化
  ```

  Of course, we need to determine some hyperparameters of the decision tree, such as the maximum depth of the tree (`'max_depth'` in the `param_grid` variable below). To determine the values of these hyperparameters, we use `GridSearchCV` in the `sklearn.model_selection` module to optimize some possible hyperparameters. Finally, the value of the hyperparameter that **makes the model training accuracy maximum is selected as the hyperparameter of the decision tree model**.

  ```
  from sklearn.model_selection import GridSearchCV
  ##对决策树进行优化
  decision_tree_optim = tree.DecisionTreeClassifier(criterion='entropy')
  param_grid = {'max_depth': [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]}
  optim_tree = GridSearchCV(decision_tree_optim, param_grid=param_grid, scoring='accuracy', cv=5, n_jobs=-1)
  optim_tree.fit(X_train,y_train)
  val_result = optim_tree.score(X_val,y_val)
  print(val_result)
  print(optim_tree.best_params_)
  print(optim_tree.best_estimator_)
  y_predict = optim_tree.predict(X_test)
  ```

- **Support Vector Machine classifier**

  We use the `SVC` module in `sklearn.svm` to create the support vector machine classifier directly.

  ```
  ###使用支持向量机评估
  from sklearn.svm import SVC
  svm_classification =SVC(probability=True)
  ```

  As the same above, we need to determine some hyperparameters of the support vector machine classifier, such as the Soft interval classifier (`'C'` in the `param_grid` variable below). To determine the values of these hyperparameters, we use `GridSearchCV` in the `sklearn.model_selection` module to optimize some possible hyperparameters. Finally, the value of the hyperparameter that **makes the model training accuracy maximum is selected as the hyperparameter of the support vector machine classifier model**.

```python
##对支持向量机进行优化
svm_optim = SVC(probability=True)
param_grid = {'C':[1,5,10]}
optim_svm = GridSearchCV(svm_optim, param_grid=param_grid, scoring='accuracy', cv=5, n_jobs=-1)
optim_svm.fit(X_train,y_train)
val_result = optim_svm.score(X_val,y_val)
print(val_result)
print(optim_svm.best_params_)
print(optim_svm.best_estimator_)
y_predict = optim_svm.predict(X_test)
```

- **K-Nearest Neighbor classifier**

  We use the `KNeighborsClassifier` module in `sklearn.neighbors` to create the K-nearest neighbor classifier directly.

```python
###使用KNN评估
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5)
```

As the same above, we need to determine some hyperparameters of the K-nearest neighbor classifier, such as the number of neighbors (`n_neighbors` in the `param_grid` variable below). To determine the values of these hyperparameters, we use `GridSearchCV` in the `sklearn.model_selection` module to optimize some possible hyperparameters. Finally, the value of the hyperparameter that **makes the model training accuracy maximum is selected as the hyperparameter of the K-nearest neighbor classifier model**.

```python
##对knn进行优化
knn_optim = KNeighborsClassifier()
param_grid = {'n_neighbors': [2,3,4,5,6,7,8,9,10,11,12]}
optim_knn = GridSearchCV(knn_optim, param_grid=param_grid, scoring='accuracy', cv=5, n_jobs=-1)
optim_knn.fit(X_train,y_train)
val_result = optim_knn.score(X_val,y_val)
print(val_result)
print(optim_knn.best_params_)
print(optim_knn.best_estimator_)
y_predict = optim_knn.predict(X_test)
```

- **Random Forest classifier**

  We use the `RandomForestClassifier` module in `sklearn.ensemble` to create the Random Forest classifier directly. Before optimizing this classifier, we make the number of decision trees (**base-level classifiers**) in the random forest to 100 and the maximum depth of the decision trees in random forest to 10.

```python
# 使用集成学习中的随机森林评估
from sklearn.ensemble import RandomForestClassifier

rfc =
RandomForestClassifier(n_estimators=100,max_depth=10,random_state=0,n_jo
bs=12)
X_train = train[:,0:-1]
y_train = train[:,-1]
X_val = val[:,0:-1]
y_val = val[:,-1]
X_test = test[:,0:-1]
rfc.fit(X_train,y_train) #用训练集数据训练模型
val_result = rfc.score(X_val,y_val)
print(val_result)
y_predict = rfc.predict(X_test)
```

As the same above, we need to determine some hyperparameters of the Random Forest classifier, such as the number of neighbors and the maximum depth of the trees in random forest(`n_neighbors` and `max_depth` in the `param_grid` variable below). To determine the values of these hyperparameters, we use `GridSearchCV` in the `sklearn.model_selection` module to optimize some possible hyperparameters. Finally, the value of the hyperparameter that **makes the model training accuracy maximum is selected as the hyperparameter of the Random Forest classifier model**.

```python
##对随机森林进行优化
rfc_optim = RandomForestClassifier(random_state=0,n_jobs=4)
param_grid = {'n_estimators': [100,200,400,800,1000],'max_depth':
[10,20,30,50]}
rfc_optim_GSCV = GridSearchCV(rfc_optim, param_grid=param_grid,
scoring='accuracy', cv=5, n_jobs=-1)
rfc_optim_GSCV.fit(X_train,y_train)
val_result = rfc_optim_GSCV.score(X_val,y_val)
print(val_result)
print(rfc_optim_GSCV.best_params_)
print(rfc_optim_GSCV.best_estimator_)
```

- **Soft-Voting Ensemble**

  We use the `VotingClassifier` module in `sklearn.ensemble` to create the soft-voting ensemble classifier directly. In this classifier, we choose the `RandomForestClassifier` ,`SVC(Support Vector Machine classifier)` and `KNeighborsClassifier` as the **base-level classifiers (estimators).** Then, we use the weighted average probabilities (soft-voting) to determine the final output of the ensemble model. It is worth noting that in this ensemble learning model, the hyperparameters of the `RandomForestClassifier` ,`SVC(Support Vector Machine classifier)` and `KNeighborsClassifier` are **based on the optimized hyperparameters results on these three separate models above**.

```python
## 使用集成学习+soft_voting评估
from sklearn import tree
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier

clf1 = 
RandomForestClassifier(n_estimators=1000,max_depth=50,random_state=0,n_jobs=
12)
clf2 = SVC(kernel='rbf', C = 10, probability=True)
clf3 = KNeighborsClassifier(n_neighbors=3)
eclf = VotingClassifier(estimators=[('rf',clf1),('svm',clf2),('knn',clf3)],
                        voting='soft',weights=[10,4,2])
```

- **Multilayer Perceptron model (MLP)**

  MLP (Multi-Layer Perceptron) is a feed-forward neural network model that maps sets of input data onto a set of appropriate outputs. The structure of MLP network can refer to the following figure:

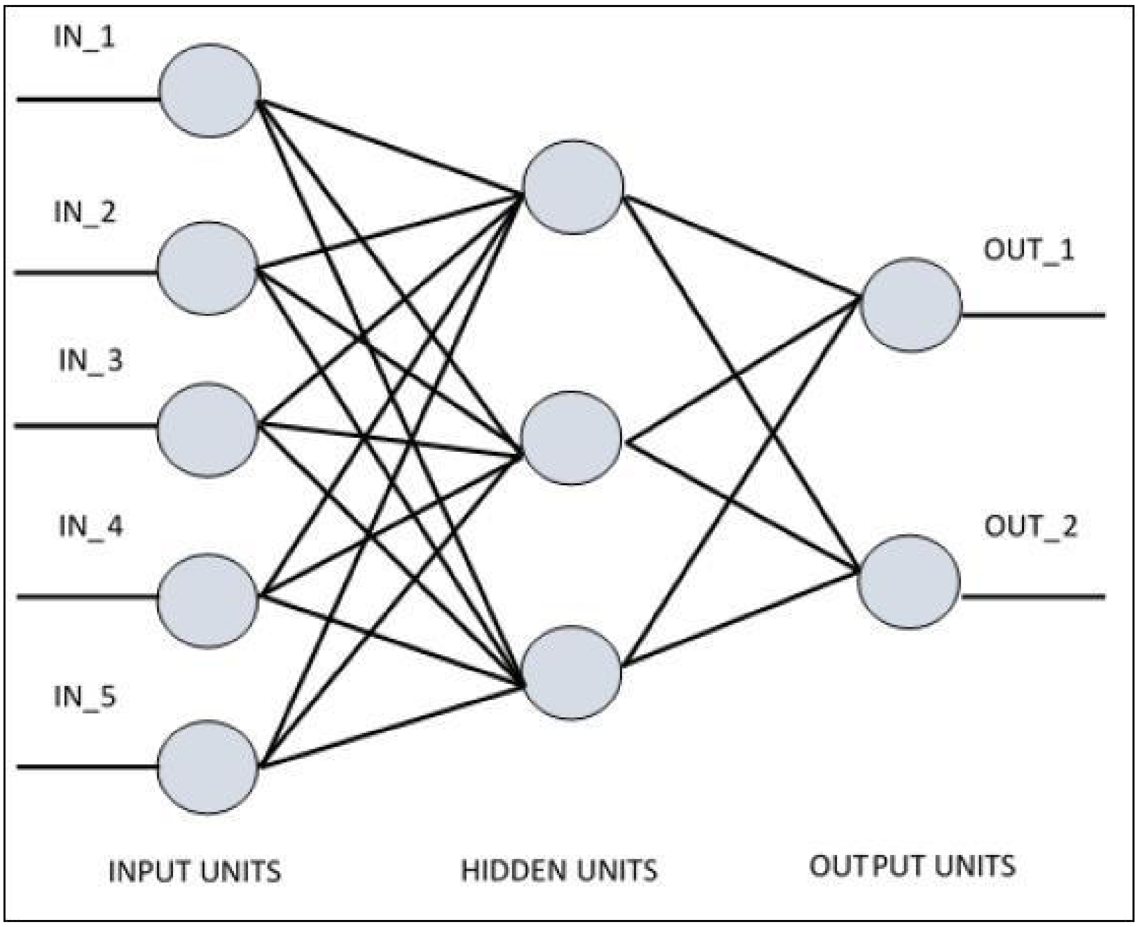

Fig.1 The structure of the multiple perceptron model

An MLP model consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable.

Since MLP is a very simple neural network structure, we will use the `Pytorch` module to implement MLP model.

```python
#MLP network setting
#device
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
class MLP_net(nn.Module):
        # fc 全连接层
        def __init__(self):
            super().__init__()
            self.fc1 = nn.Linear(33,512)
            self.fc2 = nn.Linear(512,256)
            self.fc3 = nn.Linear(256,128)
            self.fc4 = nn.Linear(128,64)
            self.fc5 = nn.Linear(64,32)
            self.fc6 = nn.Linear(32,3)

        def forward(self, x):
            x = nf.relu(self.fc1(x))
            x = nf.relu(self.fc2(x))
            x = nf.relu(self.fc3(x))
            x = nf.relu(self.fc4(x))
            x = nf.relu(self.fc5(x))
            x = self.fc6(x)
            return x
net = MLP_net().to(device)
```

With the above code, our network structure of the MLP model structure can be represented by the following figure:

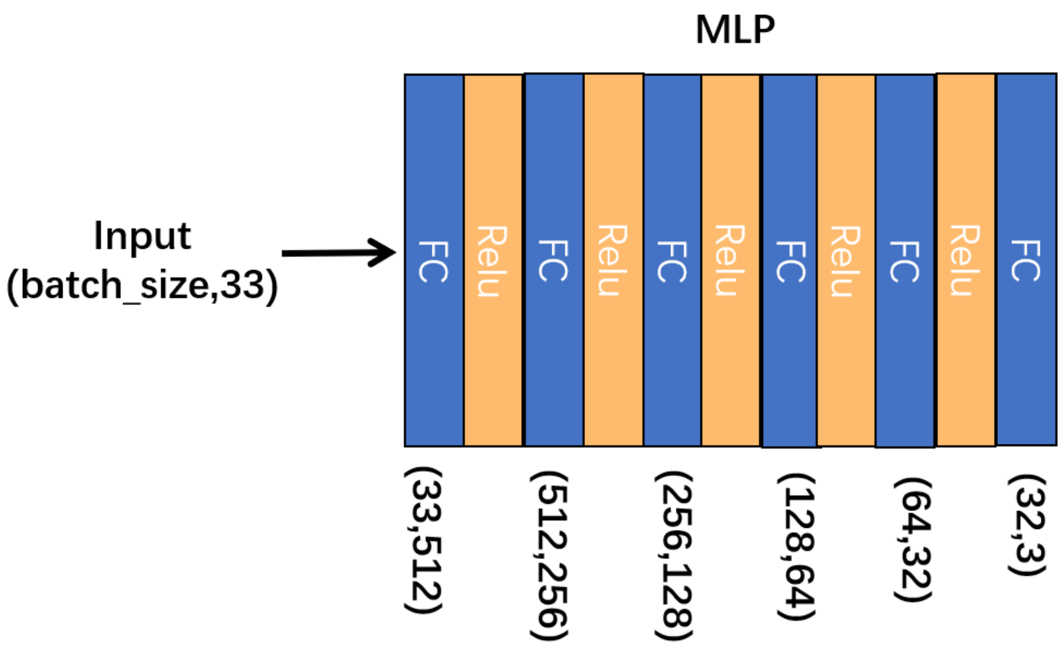

Fig.2 The network structure of the multiple perceptron model

The procedure for training and validating process of MLP model is more complex than machine learning models above, requiring us to **training and validating iteratively within a set epoch range**. Each iteration involves the following steps: `reading the data and labels in a batch`, `calculating the loss based on the output of the model and the true label`, `backpropagation (loss derivation with the model parameters)`, and `adjusting the model parameters according to the set learning rate and backpropagation results by the optimizer`. Finally, at the end of each epoch, we `adjusted the learning rate of the` `optimizer using the learning rate attenuation optimizer`. The codes of the training and validating process are shown below:

```python
def accuracy(y_hat, y):
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())

#training process
#log to record loss and acc
log=np.zeros([num_epoch,3])#train_loss,val_loss,val_accuracy
for epoch in range(st_epoch+1, num_epoch + 1):
    #training
    net.train()
    train_loss = []
    train_acc=[]
    for batch, data in enumerate(loader_train, 1):
        # forward pass
        image = data['X'].to(device)
        label = data['y'].to(device).reshape(-1)
        # print(image.shape, label.shape)
        output = net(image)
        # backward pass
        optim.zero_grad()
        loss = Loss(output, label)
        loss.backward()
        optim.step()
        train_loss += [loss.item()]
        print("TRAIN : EPOCH %04d / %04d | BATCH %04d / %04d | LOSS %.4f" % (epoch, num_epoch, batch, num_batch_train, np.mean(train_loss)))
    log[epoch-1,0]=np.mean(train_loss)
    scheduler.step()
    #validation
    net.eval()
    val_loss = []
    val_acc=[]
    for batch, data in enumerate(loader_val, 1):
        with torch.no_grad():
            # forward pass
            image = data['X'].to(device)
            label = data['y'].to(device).reshape(-1)
            output = net(image)
            loss = Loss(output, label)
            val_loss.append(loss.item())
            val_acc+=[accuracy(output,label)]
            print("VALID : EPOCH %04d / %04d | BATCH %04d / %04d | LOSS %.4f" % (epoch, num_epoch, batch, num_batch_val, np.mean(val_loss)))
    log[epoch-1,1]=np.mean(val_loss)
    log[epoch-1,2]=np.sum(val_acc)/(num_batch_val*val_batch_size)
```

Part 4: Training hyperparameters configuration

For different machine learning models, the hyperparameters configuration used in training process are as follows(**After optimization**):

- **Decision Tree**

```
criterion = 'entropy';
max_depth = 6
```

- **Support Vector Machine classifier**

```
C = 10 ;
probability = True
```

- **K-Nearest Neighbor classifier**

```
n_neighbors = 3
```

- **Random Forest classifier**

```
n_estimators=1000;
max_depth=50;
random_state=0;
n_jobs=12
```

- **Soft-Voting Ensemble**

```
estimators=
[('rf',RandomForestClassifier(n_estimators=1000,max_depth=50,random_state
=0,n_jobs=12)),('svm',SVC(kernel='rbf', C = 10, probability=True)),
('knn',KNeighborsClassifier(n_neighbors=3))];
voting = 'soft';
weights = [10,4,2]
```

- **Multilayer Perceptron model (MLP)**

```
training/validation device = 'cuda';
training batchsize = 256;
validation batchsize = 128;
learning rate = 0.0001;
number of epoch = 60;
loss function = nn.CrossEntropyLoss();
optimizer = torch.optim.AdamW(net.parameters(),lr=lr,betas=
(0.9,0.999),eps=1e-8,weight_decay=0.01,amsgrad=False);
learning rate attenuation optimizer =
 torch.optim.lr_scheduler.CosineAnnealingLR(optimizer
=optim,T_max=num_epoch,eta_min= 1e-6,last_epoch=-1);
```

Part 5: Results

- **Accuracy Comparison**

  We tested the 6 trained models selected above on the verification set, and the accuracy rate is shown in the following table:

Model	Accuracy
Decision Tree	71.28%
Support Vector Machine classifier (SVM)	74.94%
K-Nearest Neighbor classifier (KNN)	72.09%
Random Forest classifier(Ensemble)	79.22%
Soft-Voting Ensemble	78.81%
Multilayer Perceptron model (MLP)	70.55%

  From the above table, we find that the **Ensemble learning method generally has a higher accuracy**, and the **Random Forest method has the highest accuracy among all methods**. This is because **the prediction results of the Random Forest learning model integrates the prediction results of multiple base-level classifiers (decision trees), so the accuracy and robustness of recognition will be greatly improved.**

  However, we notice that the least accurate of the above methods is MLP. This is probably because **the amount or the scale of data we input is relatively small and we do not need a complex neural network model to train it, otherwise it is easy to overfit and make the neural network model perform poorly on the test set**.

- **The state in training process**

  Next, we draw the curves of the loss function and accuracy of the MLP model during training and verification process with the epoch, as shown in the figure below:

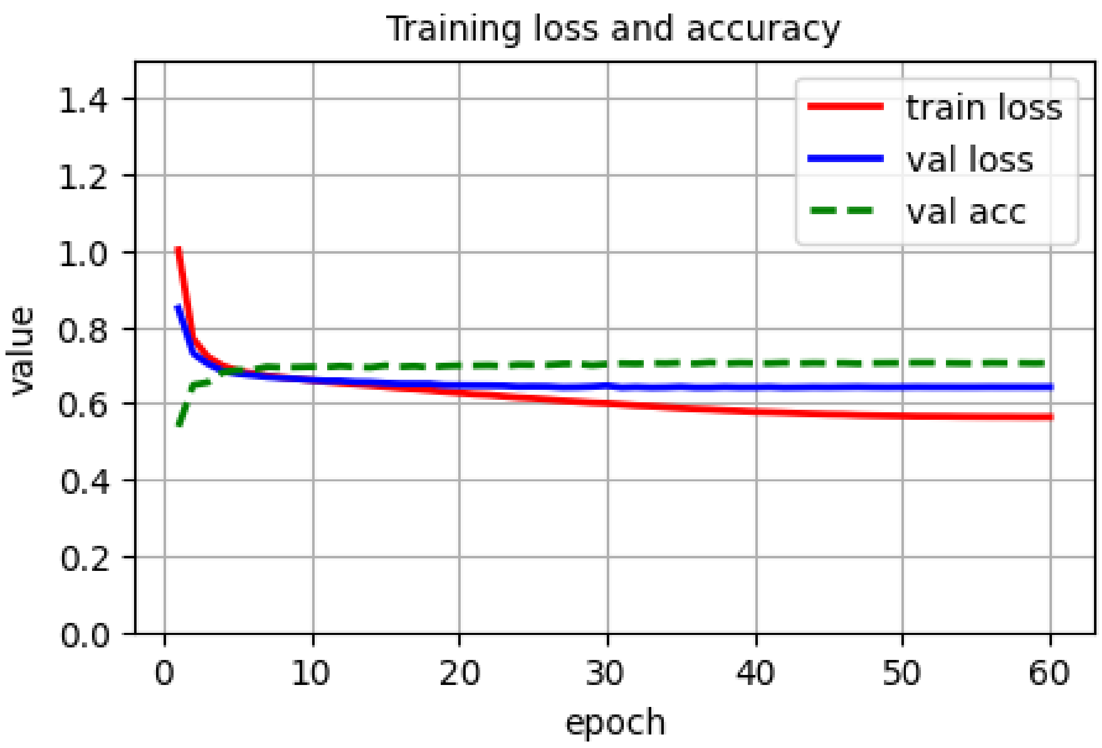

Fig.3 The loss and the accuracy during the training and verification process with the number of iterations

The figure above reflects the loss of the MLP model during the training and verification process and the accuracy during verification process with the number of iterations. It can be seen that In the initial stage of training and verification, the loss function value decreases rapidly. After 5 iteration epochs, the decline rate of the training and verification loss function starts to slow down and tends to flatten. At the same time, we can see that in the first few iteration epochs, the verification accuracy gradually increased, but around the 5 iteration epochs, the verification accuracy began to flatten. The above characteristics are consistent with the general features of MLP model during training, which reflects the rationality of our model construction.

- **The prediction results on test set**

As shown in the table above, the test result of the validation set obtained with the `Random Forest classifier` method has the highest accuracy, so we use this model to predict the test set and output the prediction results into a .csv file: `Prediction.csv`, as shown below:

```
y_predict = rfc_final.predict(X_test)
##将随机森林预测结果写入.csv文件中
with open('./Prediction.csv', 'w') as file:
    file.write("Credit_Score" + '\n')
    # 遍历列表中的元素并将其写入文件
    for item in y_predict:
        if item == 0:
            file.write('Good' + '\n')
        elif item == 1:
            file.write('Standard' + '\n')
        elif item == 2:
            file.write('Poor' + '\n')
```

Part 6: Conclusions

In this assignment, we use a variety of machine learning models to train a given dataset and test its accuracy on the validation set (we divided it manually). We found that the accuracy of ensemble learning method is higher than other methods, which indicates that ensemble learning method can improve the classification accuracy of a single base-level classifier. To our surprise, the lowest classification accuracy was obtained is using the neural network approach(MLP), suggesting that the smaller size of our dataset makes the complex neural network able to overfit the relationship between inputs and labels well during training, making it perform slightly worse on the validation set.

Finally, the data preprocessing is also very important. The key to data preprocessing in this assignment is to accurately screen out the features unrelated to the label and correctly encode the features related to the label.