Course Work Report of Artificial Intelligence

Name: Hu Zhongyang
ID: 1098496

Introduction

There are many types of classification tasks in daily lives, for example, if you want to handle the email more quickly, a good solution is to let the mailbox to filter the spams for you in advance, which the mailbox has to classify those emails into 2 classes: {Spam, not Spam}, this is called "binary-classification". Similarly, in kindergarten, children need to recognize the animals in 3 pictures and identify them into 3 classes: {Tiger, Lion, Cat}, such task is called "multi-classification".

The **purposes** of the experiment include the following:

1. Train the 5 machine learning models and use them to classify test set samples into 6 classes ({Laying, Standing, Sitting, Walking, Walking_upstairs, Walking_downstairs}).
2. Compare the classification metrics among 5 models, and analyze their performance.
3. Design and test a newly developed training set feature-concatenation method, that will transfer 561 feature columns into a single string text column (561 floats ➔ concatenate➔1 string text). We will train the transformer-based model "Bert" using this concatenated training set, and let the Bert classify the string texts into 6 classes. On doing this, we are transferring a float-sample classification problem into a text-sample classification problem (Bert can only do text classification).
4. Using A* search algorithm to search for the best hyper-parameters of Neural Network (layer numbers, hidden unit numbers, activation function categories), so that it can surpass the SVM on the performance metrics of classification.

Research Questions:

1. Can our float-feature sample classification problem transfer into a text classification problem by combining all the float features into a string and using a model called Bert to classify this string sample? (reach the same performance)
2. Which model can possibly be the best to do the multi-classification task between SVM, Neural Network, Random Forest and Bert?
3. Is there any better hyper-parameter selection policy that can help us define number of network layers and hidden unit numbers and activations in Neural Network, so that its performance can surpass SVM?

Hypothesis:

- (H1) We can use Bert to transfer multi-classification task into a text

classification problem, and Bert can reach an average performance of other models.

- (H2) SVM can reach the best performance to do the multi-classification task before A* optimize the Neural Network.
- (H3) We can use A* search algorithm to select the best hyper-parameter for the neural network, so that it can surpass SVM on classification performance (F1-score + AUC)/2.

Objectives

1. Objectives of the project

Our First Objective, is to classify each sample in the test dataset (test.cv) into 1 of the 6 classes ({Laying, Standing, Sitting, Walking, Walking_upstairs, Walking_downstairs}) using 5 different machine learning models (Neural Network, SVM, Random Forest, Bert, A* optimized Neural Network).

To do that, first we need to train our classifier, the training set is stored in the "train.csv" file that each row contains 563 columns, in those columns, the previous 561 columns are called sample feature, the last column "Activity" (col #563) is the class label, it stores a class name (e.g. Standing) that a sample assigned for.

Our second objective, is to use matplotlib to get different models' performance diagram (accuracy, training loss, cross-validation loss, precision, recall, f1-score, auc) in the form of line chart, bar chart, ROC curve, confusion matrix and classification report table.

Our third objective, is to compare all those performance metrics among all 5 models, and analyze the reason behind the metrics.

Our fourth objective, is to see whether the Bert model can be applied to the multi-classification task, and get a similar performance.

Our final objective, is to design and implement an A* algorithm that can

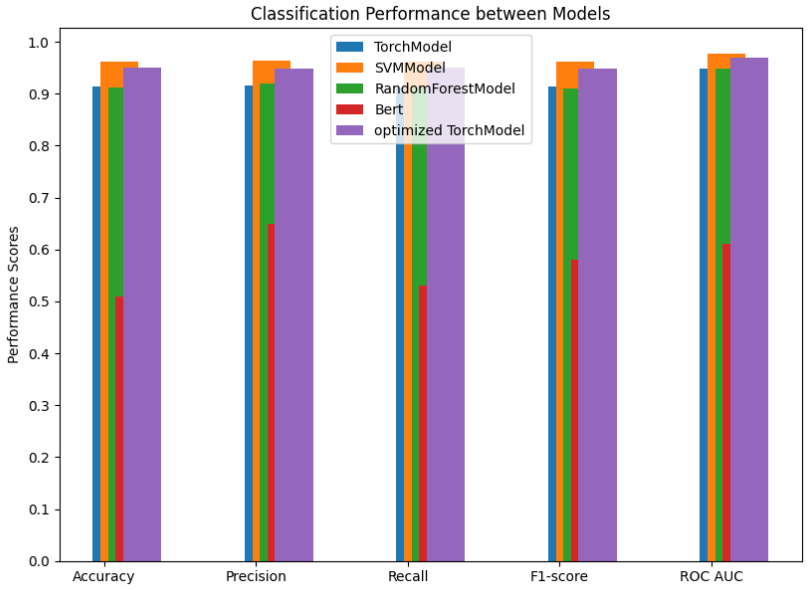
2. What the Machine Learning models aim to achieve?

1. You can consider machine learning model to be a function like $f(x) = w_1x_1 + w_2x_2 + w_3x_3$, x_1, x_2, x_3 are the features, and f(x) is the class label that the model needs to predict.
2. All 5 models will learn the pattern and the relationship (you can treat it as a function) between the training set's features and the training set's label, they will learn that through many epochs of learning process, until the model can correctly classify each sample into a class label that is exactly the same of the training set's real class label.
3. All 5 models will classify each test sample into 1 of the 6 classes. (model's prediction)

Model Selection

We choose the SVM and the A* optimized Neural Network as the top 2 best models.

If you only look at the classification performance metrics, it is obviously that SVM has the best single sore and average score of 5 metrics ([accuracy, precision, recall, f1-score, auc]), but for the A* optimized NN, the optimization effect is very obvious. Let's first talk about the SVM.




```
Accuracy        Precision       Recall  F1      AUC
model-0: [0.9331523583305056, 0.9354553261540386, 0.9326552659381768, 0.9325773376430804, 0.959650456333052]
model-1: [0.9626739056667798, 0.9630370993523759, 0.9617596395265254, 0.9621095358584367, 0.9771426678649161]
model-2: [0.9253478113335596, 0.9284364359766094, 0.9254475958664942, 0.9243163972101422, 0.9552908780340027]
model-3: [0.51, 0.65, 0.53, 0.58, 0.61]
model-4: [0.9460468272819816, 0.94534084221385, 0.9459121948927592, 0.9455012471094592, 0.9675749577615561]
```

If you look at the performance of SVM (model-1), all of the metrics are above 0.96,which is a amazing performance score.

We though it connects to the target function (lagrange method) of SVM, which can reach a local minimum in a very convenient way.

The best performance may also connect to the convex optimization which set a soft-margin constraint for the target function, it allows the model to adjust its generation ability in a floating range.

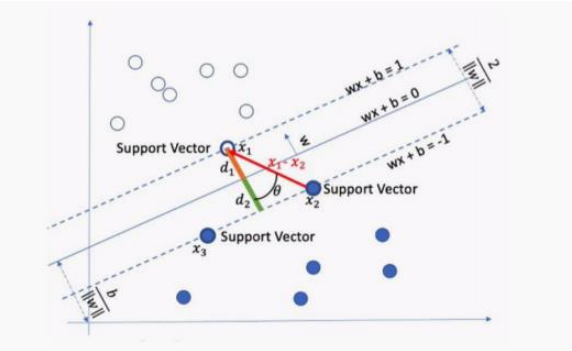
We also think it may connect to the Grid Search optimization that SVM applied, which allows it to conveniently search for the best parameter for SVM that it can reach the best performance.

1. Convex Optimization

Firstly, $w \cdot x + b = 1$ and $w \cdot x + b = -1$ are 2 hyperplanes.

We list those 2 equations together to get $\begin{matrix} w\cdot x+b\geq 1 & y=1 \\ w\cdot x+b\leq -1 & y=-1 \end{matrix}$, and combine them together, we have $y(w\cdot x+b)\geq 1$. Among those equations, $(w\cdot x+b)$ is the decision function of the SVM classifier, where w is the normal vector, x is out sample point, b is the distance between hyperplane and the origin point.

If the decision function > 1, the datapoint x should be classified as positive class (labeled as 1), otherwise, x will be labeled as -1. Also, $y(w\cdot x+b)\geq 1$ is called "constraint condition", later we will use it in lagrange method.



In addition to that, $w\cdot x+b=1$ and $w\cdot x+b=-1$ are 2 hyperplanes, so if we subtract those 2 planes, we get $w\cdot(x1-x2)=2$, and this is a vector dot product, meaning $\|w\|\cdot\|x_1-x_2\|\cdot cos(\theta)=2$, where $\|x_1-x_2\|\cdot cos(\theta)=$ d, and finally we can get $d=\frac{2}{\|w\|}$, this d is called "margin".

The objective of SVM is to maximize the margin, the larger the margin, the better that classification performance, so we need to maximize the d, which is equivalent to minimize the $\frac{1}{2}\|w\|^2$, which means $\frac{1}{2}\|w\|^2$ is our target function (objective function). So, how to minimize that? We use Lagrange method.

2. Lagrange Multiplier Method

$L(w,b,\alpha)=\frac{1}{2}\|w\|^2-\sum_{i=1}^{n}\alpha_i\left(y_i(w^Tx_i+b)-1\right)$, the first term $\frac{1}{2}\|w\|^2$ is called the regularization term (we need to minimize it), the second term is called constraint, it is used to punish the misclassified datapoints.

α_i is called Lagrange multiplier, our target is to minimize the w and b, they control the length and position of the margin. So we need to get partial derivative of w and b

and let the partial equal to 0.

$$\frac{\partial L}{\partial w} = 0 \rightarrow w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

Here, x_i is the support vector, y_i is the support vector's label, the term $\sum_{i=1}^{n} \alpha_i y_i x_i$ means n support vectors' linear combination.

$\alpha_i y_i$ is the contribution of each support vector.

$$\frac{\partial L}{\partial b} = 0 \rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0$$

So, what problem that Lagrange solved?

It finds the best hyper-plane under the constraint.

The determination of the hyperplane depends only on the non-zero Lagrange multipliers corresponding to the support vectors. This method allows SVM to effectively classify in high-dimensional space.

3. Constraints

$y_i(w \cdot x_i + b) \geq 1 - \varepsilon_i, \varepsilon_i \geq 0$

This is the constraint of the soft-margin SVM. In hard margin SVM, we hope all the datapoints to be correctly classified, which they all should satisfy $y(w \cdot x + b) \geq 1$. However, in the real world, dataset not always linear separable due to noises, so we need to import a "Slack Variable" ϵ_i to solve this problem. The slack variable ϵ_i allows some points to violate the hard-margin, which means allows those points to be inside the margin, or to be classified to the wrong side.

If $\epsilon_i = 0$: classified correctly and outside the margin.

If $\epsilon_i > 0$: classified wrongly or inside the margin.

4. Optimized target function:

$\min \frac{\| w \|^2}{2} + C \sum_{i=1}^{n} \varepsilon_i$, $\frac{1}{2} \| w \|^2$ is a regularization term, $C \sum_{i=1}^{n} \varepsilon_i$ is a loss term, the purpose of this term is to reduce the number of misclassified points. C is a positive hyper-parameter, its purpose is also to reduce the number of misclassified points.

This is also to reduce the number of points that have a looser constraint: $y_i(w \cdot x_i + b) \geq 1 - \varepsilon_i$.

5. Kernel Function

1. The objective of the kernel function:
   Mapping the sample vector from low-dimension feature space to high-dimension feature space.

2. Why do we need kernel function to calculate the dot-product of X1 and X2?
   This allows the SVM algorithm to find linear decision boundaries in high-dimensional space.

3. **Gaussian Kernel (also called RBF, Radial Basis Function):**
   Objective: It measures the similarity between two sample points by calculating the exponential decay of their Euclidean distance.

$$K(X_i, X_j) = e^{-\frac{\| X_i - X_j \|^2}{2\sigma^2}}$$

   (σ) controls the width of the function, $(\gamma) = \frac{1}{2\sigma^2}$ also controls the width of the function. Be aware of that, this function do not need any prior knowledge about the data.

4. **Polynomial Kernel:**
   Objective: It calculates the dot product of data points, and map the result into a polynomial space to measure the similarity between vectors.
   Be aware: h is the polynomial degree. The operation of adding one ensures that even when all original features are zero, there can still be non-zero kernel output.

$$K(X_i, X_j) = (X_i \cdot X_j + 1)^h$$

6. SVM code

```python
class SVMModel:
    def __init__(self):
        # kernel function list
        # sigma: 控制高斯核函数的宽度， sigma越小， 那么函数值随两点间（X_i, X_j)距离的增大而减小地越快
        # gamma: gamma = 1/(2*sigma^2)
        # C: 在多项式核函数中，确保了即使在原始特征全部为零时，也能有非零的核函数输出
        self.kernel=[
                    {
                        'kernel': ['rbf'],
                        'gamma': [1e-3, 1e-4],
                        'C': [10, 100]
                     },
                    {
                        'kernel': ['linear'],
                        'C': [10, 100]
                        }
                    ]
        # build a primary SVM model
        # SVC: SVM classifier
        # cv: number of folds in cross-validation

        # self.model = GridSearchCV(SVC(probability=True),self.kernel, cv = 5)
        self.model = GridSearchCV(SVC(),self.kernel, cv = 5)

```

```python
    def set_kernel(self, kernel):
        '''
        设置核函数列表
        '''
        self.kernel = kernel

    def set_kernel_params(self, kernel_name, gamma, C):
        '''
        设置核函数的参数，比如：
        1. 多项式核的 h和c
        2. rbf和的sigma， 8它控制着核函数的宽度， 影响模型的灵敏度
        '''
        kernel_index = -1

        # search kernel function in the kernel list
        kernel_list = self.kernel
        for i, k_dict in enumerate(kernel_list):
            if kernel_list['kernel'] == kernel_name:
                kernel_index = i

        if kernel_index ==-1:
            return False
        else:
            kernel_list[kernel_index][kernel_name]['gamma'] = gamma
            kernel_list[kernel_index][kernel_name]['C'] = C
            return True
```

```
    def forward(self, x, y= None):
        '''
        generate model's prediction
        '''

    def __fit(self, X_train_scaled, Y_train):
        '''
        fit the training data
        '''
        self.model.fit(X_train_scaled, Y_train)


    def evaluate(self, X_train_scaled, Y_train):

        print(" =======  It needs about 5 minutes, Please be patient ~~~ ====================")
        self.__fit(X_train_scaled, Y_train)
        print(f'Best score for training data:{self.model.best_score_}')

        print(f'Best C: {self.model.best_estimator_.C}')
        print(f'Best Kernel: {self.model.best_estimator_.kernel}')
        print(f'Best Gamma: {self.model.best_estimator_.gamma}')

        final_model = self.model.best_estimator_
        return final_model
```

The above 3 snippets of code belong to a self-defined SVM class called "SVMModel", in this class, the content in the "__init__" is the core of the entire SVM model.

"self.kernel" is a kernel function list, it stores 2 kernel function dictionaries, the first one is the Gaussian Kernel Function (RBF), which is this one: $K(X_i, X_j) = e^{-\frac{\| X_i - X_j \|^2}{2\sigma^2}}$, gamma γ is the width of the kernel, it controls how fast the function value will decrease as the distance $< X_i, X_j >$ increases. We have 2 gamma values: {1e-3, 1e-4}, later, the Grid Search optimization will pick the best parameter gamma for us.

Also, the key "C" in the 2 dictionaries is the "Regularization coefficient" , it is a parameter in the soft-margin SVM's target function $\min \frac{\| \mathrm{w} \|^2}{2} + C \sum_{i=1}^{n} \varepsilon_i$, which is used to control the penalty level of the margin violation (inside the margin, or classified to the other side).

If we increase the value of C, enlarging the penalty, there will be less model classification, but maybe will also generate overfitting problem.

So, we need a very careful selection of C.

However, do not worry, this task is handed over to the Grid Search optimization.

The second kernel function is called linear kernel, we can only adjust parameter C in it. This kernel is very useful when the data is linearly separable.

7. What is Grid Search?

It is used to systematically traverse different parameter combination (e.g. gamma = 1e-4, C=10), and use cross-validation to calculate the performance of this

combination and finally settle the best parameter combination.

The Grid Search will firstly define a multi-parameter grid (matrix). Assuming we have 3 rows in a grid, each row represents a certain type of parameter (e.g. first row is kernel type, second row is gamma, third row is C). For each combination of 3 rows, the Grid Search will calculate many different scored using cross-validation, and get the following result:

will be represented by a `cv_results_` dict of:

```
{
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...),
'split0_test_score'  : [0.80, 0.70, 0.80, 0.93],
'split1_test_score'  : [0.82, 0.50, 0.70, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.85],
'std_test_score'     : [0.01, 0.10, 0.05, 0.08],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.80, 0.92, 0.70, 0.93],
'split1_train_score' : [0.82, 0.55, 0.70, 0.87],
'mean_train_score'   : [0.81, 0.74, 0.70, 0.90],
'std_train_score'    : [0.01, 0.19, 0.00, 0.03],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.01, 0.06, 0.04, 0.04],
'std_score_time'     : [0.00, 0.00, 0.00, 0.01],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
}
```

The key is, for each parameter combination, Grid Search splits the dataset into several parts (usually k parts) and performs k-fold cross-validation.

According to the above explanation, we think SVM is the best to handle such multi-classification task because it has a good target function optimization solution, as well as a very good model parameter search solution.

A* Optimized Neural Network

So, next, under the inspiration of Grid Search, we also wonder that whether the Neural Network can adopt same procedure to enhance its performance until it can surpass the SVM. Do we decide to use A* search algorithm to find the best hyper-parameters of the Neural Network.

First, we define the hyper-parameters of NN to be: {number of linear layer, numbers of hidden units, types of activation function}. We change our NN model class to be the following, so that we can get different Neural network object with different param

```python
# self-defined neural network class
class TorchModel(nn.Module):
    def __init__(self, input_size: int=561, layers:int=4, units:int=15, hidden_activation:nn.Module=nn.Tanh):
        '''
        inputsize: 输入样本的特征数量
        '''
        super(TorchModel,self).__init__() # 调用父类构造器初始化神经网络

        # 线性层的输出节点个数 == 分类的类别数(6)
        # input_size 就是每一个样本的特征数量 (假设是20)
        # 假设我们有一个输入矩阵 X(row=n, col = 20), 一个线性层的权重矩阵W(row = 20, col = 6)
        # 线性层的作用就是计算 X*W +b, 其中X*W(row = n, col = 6), b是常数（偏置值）
        self.linear_list=nn.ModuleList()

        self.hidden_activation = hidden_activation
        if layers<1:
            linear1 = nn.Linear(input_size, units)
            self.linear_list.append1(linear1)
        else:
            linear1 = nn.Linear(input_size, units)
            self.linear_list.append(linear1)

            for i in range(1,layers-1):
                linear2 = nn.Linear(units, units)
                self.linear_list.append(linear2)

            linear3 = nn.Linear(units, 6)
            self.linear_list.append(linear3)

        # 线性层输出矩阵的每一行都是一个分类向量，形如 [1.2, 3.4, 5.5 ,0.2, 0.8, 0.9]
        # 这些数乍一看是无规则的，没什么意义， 因此我们要用softmax函数给他过滤一下
        # 就变成了 [0.2, 0.2, 0.1, 0.05, 0.05, 0.4], 这就是一个概率分布，0.4是最大的概率，因此这个样本被分到第6类
        self.activation = nn.Softmax(dim=1) # dim表示应用softmax的维度， dim=1表示对第二维， 也就是列应用softmax
        self.loss = nn.CrossEntropyLoss() # 交叉熵损失函数
```

store all the linear layers

all hidden layers' activation

input layer

hidden layers

output layer

```python
    def forward(self, x:torch.Tensor, y:torch.Tensor=None):
        '''
        x: 我们输入的样本矩阵
        y: 样本的真实标签， 如果y给出了，我们需要返回loss， 如果
            没给出，我们直接返回模型预测的y_pred
        '''

        # 得到线性层的输出
        linear_list = self.linear_list

        for i, ll in enumerate(linear_list):
            x = ll(x)
            if i!=len(linear_list)-1:
                x =self.hidden_activation()(x) # 过一下线性层的激活函数

        # 放入激活函数得到6个类的概率分布
        # softmax做完以后， 结果会被自动放入one-hot 函数，转为 0-1矩阵
        # 为什么要用one-hot, 因为单纯的概率分布向量看着不是很直观，并且loss不是太好计算
        # [0.2, 0.2, 0.1, 0.05, 0.05, 0.4]  ===one-hot==> [0, 0, 0, 0, 0, 1]  ----- 代表分到了第6类
        # [0, 0, 1, 0, 0, 0]  ----- 代表分到了第三类

        # y_pred.shape = (n, 6)
        y_pred:torch.Tensor= self.activation(x)



        if y is not None:
            # 计算预测值和真实值之间的损失并返回

            y=y.long() # 转为LongTensor, 这是强制要求
            # 注意！！！ 预测值必须在前， 真实值必须在后，否则报错
            return self.loss(y_pred, y)
        else:
            # 直接返回预测值
            return y_pred
```

After that, we define our A* algorithm.

Firstly, we define a Node class for each node in A*, and use a priority queue to store all the explored nodes (frontier nodes).

```python
# 优先队列
import heapq

class Node:
    def __init__(self, layers, units, activation, performance:float=0):
        # hidden layer numbers
        self.layers = layers
        # hidden units numbers
        self.units = units
        # nn.Relu ...
        self.activation:nn.Module = activation
        # avg(f1+auc)
        self.performance:float = performance
        # 父节点指针
        self.parent:Node = None
        # 从开始在当前节点的路径上的节点数 (包括当前)
        self.path_node_num: int = None

        # cost(start, current)
        self.cost:float=0

    def __lt__(self, other):
        return self.performance < other.performance
```

We then find a method of how to search the neighbors of a node:

```python
def get_neighbors(node:Node):
    # Generate neighbors by changing one hyperparameter at a time
    neighbors = []

    # generate neighbor by changing NN's layer number
    # 减少、或增加层数的节点都可以被当做邻居
    if node.layers > 1:
        neighbors.append(Node(node.layers - 1, node.units, node.activation, 0))
    neighbors.append(Node(layers=node.layers+1, units=node.units, activation=node.activation, performance=0))


    # Generate neighbors by changing the number of units
    if node.units > 1:
        neighbors.append(Node(node.layers, node.units - 1, node.activation, 0))
    neighbors.append(Node(node.layers, node.units + 1, node.activation, 0))


    # Generate neighbors by changing the activation function
    activation_functions = [nn.ReLU, nn.Sigmoid, nn.Tanh]
    current_activation_index = activation_functions.index(node.activation)
    next_activation_index = (current_activation_index + 1) % len(activation_functions)
    neighbors.append(Node(node.layers, node.units, activation_functions[next_activation_index], 0))

    return neighbors
```

After that, we designed our evaluation function f(n), and consider it as each NN model node's performance. Since $f(n) = c(n) + h(n)$ for any node n, so we still need to define the cost function c(n) and heuristic function h(n). To be noticed, we set the evaluation value to be the inverse of the average of 1/h(n) and 1/c(n): $f(n) = \dfrac{1}{\dfrac{\dfrac{1}{cost} + \dfrac{1}{heuristic}}{2}}$, because f(n) value is "the smaller, the better", but our performance for example f1 = 0.96, is "the larger the better". So we decide to let the a single node's cost = $\dfrac{1}{performance}, where, performance = \dfrac{f1 + auc}{2}$, however, in A*, cost must also take the history cost into consideration, which means we need to add the

parent's cost to the current node's cost, however, that is much troublesome, so we let the cost = $\frac{1}{\frac{1}{\frac{1}{parent\_cost}+current\_cost}{2}}$ to make it simpler, but the cost converges much more slower.

Also, we did the same to the heuristic function, normally we will consider the Euclidean distance between the current node and the goal node, but in our case, we still needs the "Inverse of Average", which $h(n)=\frac{1}{\frac{1}{\frac{1}{currentNode\_performance}+goal}{2}}$.

However, it also converges very slowly, which cause the $f(n)$, the evaluation function also converges very slowly.

This means, if we set up a very high goal for A*, it will be very hard to get there, and takes a very long times (few hours), and maybe never get there.

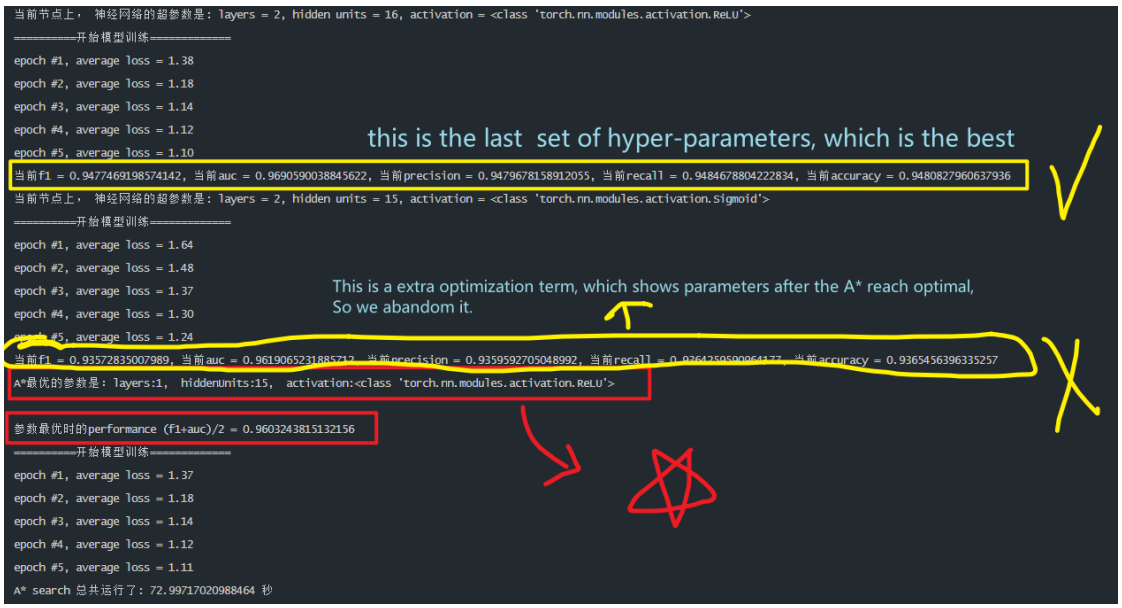
```
175    '''
176    以下这段是周三pre完后加的
177    '''
178    def evaluate_node(node:Node, start:Node, goal:float,
179                      train_x:torch.Tensor, train_y:torch.Tensor,
180                      test_x:torch.Tensor, test_y:torch.Tensor, test_y_label:np.ndarray, encoder:LabelEncoder):
181        # Train and evaluate a neural network with the given hyperparameters
182        # g = 平均性能的倒数
183        g_value = cost(start, node, train_x, train_y, test_x, test_y, test_y_label, encoder)
184        # h = 平均性能的倒数
185        h_value = heuristic(node, goal)
186        # f是前面两个平均性能的倒数
187        f_value = 1/((1/g_value + 1/h_value)/2)
188
189        return f_value
```

```
191    def heuristic(node:Node, goal:float):
192
193        if node.performance ==0:
194            return 1/goal
195        # 取当前节点性能与目标节点性能平均的倒数
196        return 1/((1/node.performance+goal)/2)
```

```python
'''
以下这段是周三pre完后加的
'''
def cost(start:Node, node:Node, train_x:torch.Tensor, train_y:torch.Tensor,
         test_x:torch.Tensor, test_y:torch.Tensor, test_y_label:np.ndarray, encoder:LabelEncoder):

    model = TorchModel(
        input_size=train_x.shape[1], layers=node.layers, units=node.units, hidden_activation=node.activation)

    print(f'当前节点上， 神经网络的超参数是: layers = {node.layers}, hidden units = {node.units}, activation = {node.activation}')

    trained_model = train_model(model, train_x, train_y)

    # 获得预测值
    test_x=test_x.detach()
    y_pred:torch.Tensor= trained_model(test_x)
    y_pred=y_pred.detach() # 2947 x 6
    y_pred = torch.argmax(y_pred, dim=1).detach()
    # 转为string标签
    y_pred_label = encoder.inverse_transform(y_pred)

    # 计算f1
    f1 = f1_score(test_y_label, y_pred_label, average='macro')

    # 计算auc
    auc = roc_auc_score(label_binarize(test_y, classes=[0,1,2,3,4,5]), label_binarize(y_pred, classes=[0,1,2,3,4,5]), multi_class='ovr')


    accuracy = accuracy_score(test_y, y_pred)

    precision = precision_score(test_y, y_pred, average= 'macro')

    recall = recall_score(test_y, y_pred, average= 'macro')


    print(f'当前f1 = {f1}, 当前auc = {auc}, 当前precision = {precision}, 当前recall = {recall}, 当前accuracy = {accuracy}')
```

```python

    # 当前模型的性能 (为了缩短优化时间，我们只比较两项关键指标的均值)
    node_cost = (f1+auc)/2

    # 当前真实代价(cost)和父节点代价做一个平均 ==> 模拟从原点到当前节点的代价
    # 取倒数是因为我们比的是谁的代价更小
    if node.parent is not None:
        avg_cost = ((1/node.parent.cost) + node_cost)/2
    else:
        avg_cost = node_cost

    # 节点的最终cost == 节点所在的<start, node>路径上的平均性能
    # 为什么要用倒数， 因为cost比较的是谁的值更小
    node.cost = 1/avg_cost

    return node.cost

def is_goal(node:Node):
    # Check if the performance of the node is good enough

    '''
    我们这里将f1-score 和 auc的平均值设为性能指标
    '''
    goal = 0.96

    if node.performance == 0:
        return False
    return (1/node.performance) >= goal
```

After few rounds of testing, we find that to set the goal of the A* to be $\frac{f1+auc}{2}=0.96$ shall be a better solution, since 0.975 is a goal that A* can never reach.



You can see the result of the A* optimization, it finds the best hyper-parameter of neural Network to be {layer number = 1, hidden units = 15, activation function = ReLU}, and the best performance is $\frac{f1+auc}{2}=0.9603$. This is a huge improvement (0.04) compare to the original 0.92. This why we select the A* optimized NN as one of the 2 best models, because its improvement on performance is really huge.

Results

In the data preprocessing stage, the previous 4 models (NN, SVM, RF, optimized-NN) are nearly the same:

```
36  def build_dataset()->pd.DataFrame:
37
38      '''
39      create training set and test set
40      '''
41      # shuffle the sample order in the dataset
42      train: pd.DataFrame = pd.DataFrame(shuffle(pd.read_csv('./train.csv')))
43      test: pd.DataFrame = pd.DataFrame(shuffle(pd.read_csv('./test.csv')))
44
45      # test missing value
46      # isnull: Return a dataFrame of type bool
47      # values: convert dataframe into numpy
48      # any(): Return True if there are missing values
49      print("Does train has any missing value? %s"%train.isnull().values.any())
50
51      # Fill in missing values
52      if train.isnull().values.any():
53          # Fill in the missing values of each column with the average value of each column,
54          # inplace: modify in situ
55          train.fillna(train.mean(), inplace=True)
56          test.fillna(test.mean(), inplace=True)
```

(Figure-1)

In Figure-1, we first shuffle all the samples in the dataset, and then we detect missing values in each row, if there is one, that row will return a True, and finally the entire dataset will return a Boolean vector that contains True or False (isnull()), after that we use “any()” to detect whether any “True” exists, if it does, we will fill the missing value.

```
59    print("===== 数据处理有点慢，  请耐心等待 ==========")
60
61
62    # Separate feature columns and label columns in the dataset
63    X_train = pd.DataFrame(train.drop(['Activity','subject'],axis=1))
64    # values: Get the value of certain DataFrame column, and transfer it into numpy
65    # astype(object): 将列中的值转为object类型
66    Y_train_label = train.Activity.values.astype(object)
67
68
69    X_test = pd.DataFrame(test.drop(['Activity','subject'],axis=1))
70    Y_test_label = test.Activity.values.astype(object)
```

(Figure-2)

In Figure-2, we separate the features columns and the target label column apart.

```
56    # dataset最后一列类标签转为数字0-5
57    encoder = LabelEncoder()
58    encoder.fit(Y_train_label)
59    Y_train = encoder.transform(Y_train_label)
60
61    encoder.fit(Y_test_label)
62    # get am array that each ele is a class-mapping-number [0-5]
63    Y_test = encoder.transform(Y_test_label)
64
65    # 里面存了针对Y_test_label的编码器
66    final_encoder = encoder
```

(Figure-3)

In Figure-3, we use the LabelEncoder to transfer a class-string-label column Y_train ([Laying, Standing, Sitting, Walking, Walking_upstairs, Walking_downstairs]) into a class-integer-label column ([0, 1, 2, 3, 4, 5]).

```
73    # normalize every values in feature columns
74    scaler = StandardScaler()
75    X_train_scaled = scaler.fit_transform(X_train)
76    X_test_scaled = scaler.fit_transform(X_test)
77
78
79    Y_train = np.array(Y_train)
80    Y_test = np.array(Y_test)
```

(Figure-4)

In Figure-4, we use the StandardScaler to normalize all the feature columns in the dataset using the formula $\frac{x-\mu}{\sigma}$, where x is the value under that feature, μ and σ are mean and std of that feature column.

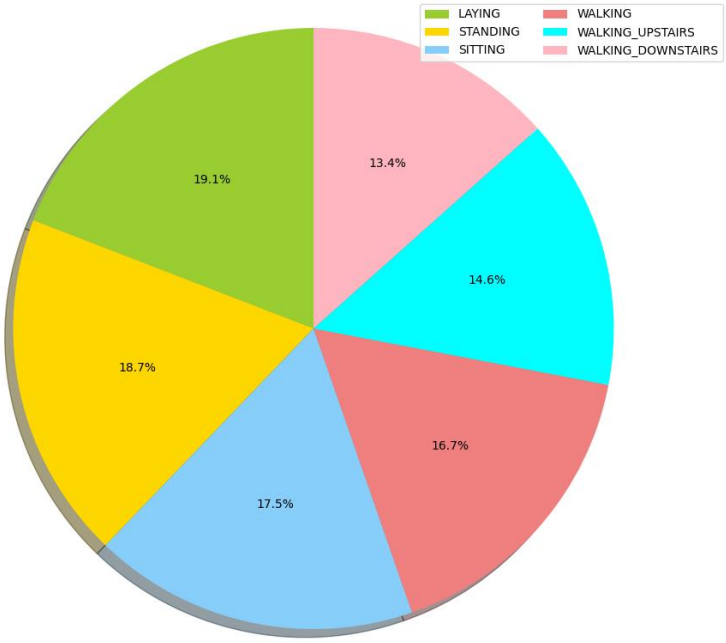
```
232   # 将所有数据集全部转为Tensor
233   X_train = torch.FloatTensor(X_train_scaled)
234   Y_train = torch.FloatTensor(Y_train)
235   X_test = torch.FloatTensor(X_test_scaled)
236   Y_test = torch.FloatTensor(Y_test)
```

(Figure-5)

In Figure-5, for Torch framework, all the model input dataset should be Tensor (A

high-dimensional matrix), so we need to do the conversion.

After using the pyplot to draw the pie chart of the class distribution in the training set, we get the following diagram:



(Figure-6)

In addition to that, we also print the shape of the training set and the testing set:

```
Does train has any missing value? False
===== 数据处理有点慢， 请耐心等待 =========
X_train.shape: torch.Size([7352, 561])
Y_train.shape: torch.Size([7352])
X_test.shape: torch.Size([2947, 561])
Y_test.shape: torch.Size([2947])
--------------------------------
Number of feature = 561
```

(Figure-7)

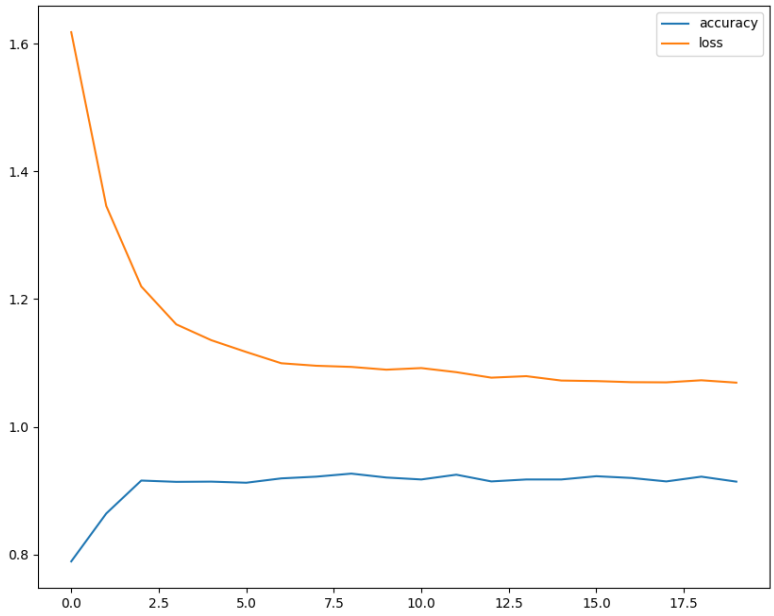
1. Run the Neural Network training and evaluation process:

Firstly, the Neural Network model is stored in the file “activity_classifier_torch.py”, and we sometimes treat NN as the “TorchModel”.

```
分类的正确率为：0.9144893111638955
epoch #14, average loss = 1.08
分类的正确率为：0.9175432643366135
epoch #15, average loss = 1.07
分类的正确率为：0.9175432643366135
epoch #16, average loss = 1.07
分类的正确率为：0.9226331862911435
epoch #17, average loss = 1.07
分类的正确率为：0.9199185612487275
epoch #18, average loss = 1.07
分类的正确率为：0.9144893111638955
epoch #19, average loss = 1.07
分类的正确率为：0.9219545300305395
epoch #20, average loss = 1.07
分类的正确率为：0.9141499830335935
```

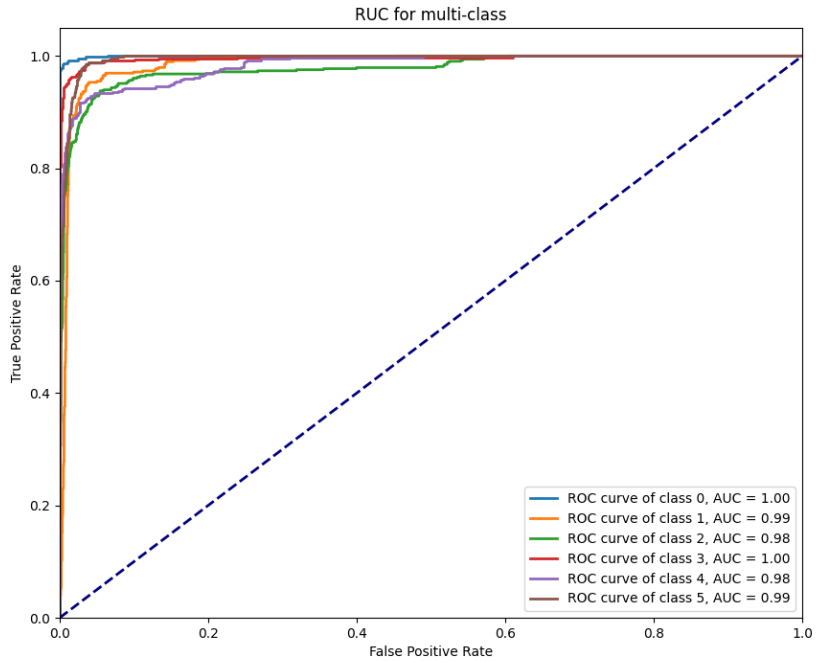
(Figure-8)

Above is NN's training loss and training accuracy during 20 epochs.



(Figure-9)

Figure-9 shows the tendency of loss and accuracy during the 20 rounds training, it shows, as the loss decreases, the accuracy will increase and close to 1, converging around 0.92.



(Figure-10)

The ROC curves in Figure-10 are generated using the following code:

```python
def get_roc(test_y:torch.Tensor,test_y_pred:torch.Tensor):
    '''
    test_y: testing set'class label vector
    test_y_pred: 测试集的标签对应的概率分布矩阵 shape(n, 6)

    The meaning of One-vs-Rest policy

    It is a commonly used method in multi classification problems,
    which decomposes the multi classification problem into multiple binary classification problems to solve.

    Specifically, for a multi classification problem with N categories, a one-vs-rest strategy will create a binary classification model for each category,
    Each model considers this category as positive and all other categories as negative.
    In this way, we can obtain N binary classification models, each of which can distinguish a specific category from all other categories.
    Based on this strategy, we can draw N ROC curves.

    '''

    import warnings
    from sklearn.exceptions import UndefinedMetricWarning

    # 忽略 UndefinedMetricWarning, 用于忽略控制台的警告信息
    warnings.filterwarnings("ignore", category=UndefinedMetricWarning)


    # 准备画布
    plt.figure(figsize=(10, 8))

    from sklearn.preprocessing import label_binarize
```

(Figure-11)

```
    # 为每个类别绘制ROC曲线
    for i in range(test_y_pred.shape[1]):
        # test_y_pred[:, i] 是一个一维数组，表示模型预测每个样本属于第 i 类的概率。
        # pos_label 参数在 roc_curve 函数中用于定义哪个类别被视为正类。
        # 将 test_y_bin[:, i], test_y_pred[:, i] 结合起来可以计算 TPR 和 FPR
        y=test_y.detach().numpy()
        y_prob = test_y_pred
        y_prob = y_prob[:, i].detach().numpy()









        fpr, tpr, thresholds = roc_curve(y, y_prob, pos_label=i)
        # 计算auc的值
        roc_auc = auc(fpr, tpr)
        # lw: linewidth
        plt.plot(fpr, tpr, lw = 2, label = 'ROC curve of class %d, AUC = %0.2f'%(i, roc_auc))

    # 添加对角线
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

    # 设置图的其他属性
    # 将 x 轴的显示范围设置为从 0.0 到 1.0
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('RUC for multi-class')
    plt.legend(loc="lower right")
    plt.show()
```

(Figure-12)

It is very clear that, we draw a ROC curve for each of the 6 classes, we use the One-vs-Rest policy to build a binary-classifier for each of the 6 classes, for example, if we are drawing the roc for class “Standing”, we will take it as the positive class and all other 5 classes as the negative classes.

In this “get_roc(test_y, test_y_pred)” function, test_y is the real class-number label in the test set ([0,1, 2, 3, 4, 5]), and the test_y_pred is a matrix that each line is a sample’s probability distribution on 6 classes (e.g. [0.1, 0.2, 0.1, 0.5, 0.05, 0.05], sum together =1 because of softmax). For each class “i”, we get the i^{th} column of the test_y_pred matrix (all sample’s probability on class i) and the real class-number label vector test_y, to calculate the TPR (True Positive Rate) and FPR for the ROC.

So, how to identify performance as good or bad in ROC?

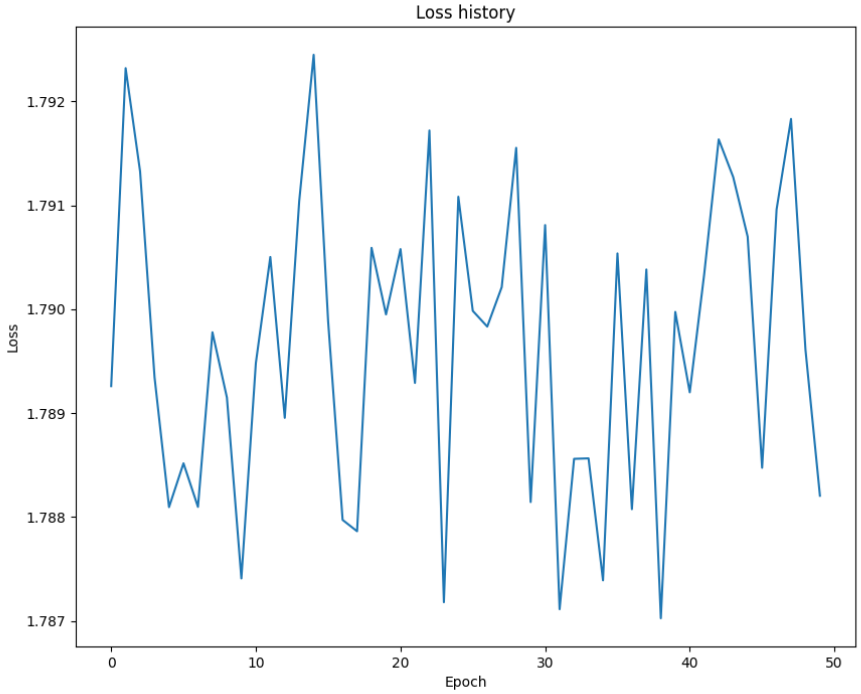
Simple, just look at the curve, if the ROC curve is closer to the upper left corner, its classification performance will be better.

For example, in Figure-10, we see the blue curve of class 0 (Laying) is closer to the upper left corner, and the green curve of class 2 (Standing) is not very close to the corner, we can say that the class Laying’s classification performance is better that the class Standing’s.

For another way, you can just check the AUC (area under curve) value, you can see the class Laying’s AUC = 1.0, higher that class Standing’s, which equals to 0.98.

We also write a K-fold cross validation function:

```python
def k_fold_cross_validation(k):
    '''
    我们将训练集分为K折，其中K-1份组成训练集，剩下一份是测试集，轮流跑K次，最后取平均测试结果
    '''
    input_size = 561  # 输入向量维度
    learning_rate = 0.001  # 学习率

    model = TorchModel(input_size=input_size)

    # 创建优化器
    optim = torch.optim.Adam(params = model.parameters(), lr=learning_rate)

    dataset:list[list] = shuffle(pd.read_csv('./train.csv'))

    dataset: pd.DataFrame = pd.DataFrame(dataset)

    # 分离特征和标签， 以及类别映射
    X_dataset = pd.DataFrame(dataset.drop(['Activity','subject'], axis=1))
    X_dataset = torch.tensor(X_dataset.values, dtype = torch.float32)

    Y_dataset_label = dataset.Activity.values.astype(object)
    labelEncoder = LabelEncoder()
    Y_dataset: np.ndarray =labelEncoder.fit_transform(Y_dataset_label)
    Y_dataset = torch.tensor(Y_dataset, dtype = torch.long)


    # dataset=np.array(dataset)

    # print(dataset)
    dataset_len = len(dataset)
    fold_size = dataset_len // k

    # 记录每一轮，测试集上的损失
    watch_loss= []

    # 在开始交叉验证之前保存模型的初始参数
    import copy
    initial_state_dict = copy.deepcopy(model.state_dict())

    for i in range(k): # k轮交叉验证

        # 在每次迭代开始时加载初始参数
        model.load_state_dict(copy.deepcopy(initial_state_dict))

        X_test = X_dataset[i*(fold_size):(i+1)*fold_size]
        Y_test = Y_dataset[i*(fold_size):(i+1)*fold_size]

        X_train = torch.cat((X_dataset[:i*fold_size], X_dataset[(i+1)*fold_size:]), dim=0)
        Y_train = torch.cat((Y_dataset[:i*fold_size], Y_dataset[(i+1)*fold_size:]), dim=0)
        # X_train = X_dataset[(i+1)*fold_size:]
        # Y_train = Y_dataset[(i+1)*fold_size:]

        loss = model(X_train, Y_train)
        loss.backward()  # 计算梯度
        optim.step() # 更新参数
        model.zero_grad() # 梯度归零， 每一个批次只能用该批次的损失函数来计算梯度

        # 测试集的误差
        loss_test = model(X_test, Y_test)
        watch_loss.append(loss_test.item())

        print(f'{k}-fold round # {i+1}, loss = {loss_test.item():.2f}')
        # acc=evaluate(model, test_x, test_y)
        # log.append([acc,np.mean(watch_loss)])

    print(f'{k}-fold CV\'s average loss = {np.mean(watch_loss):.2f}')
```



(Figure-13)

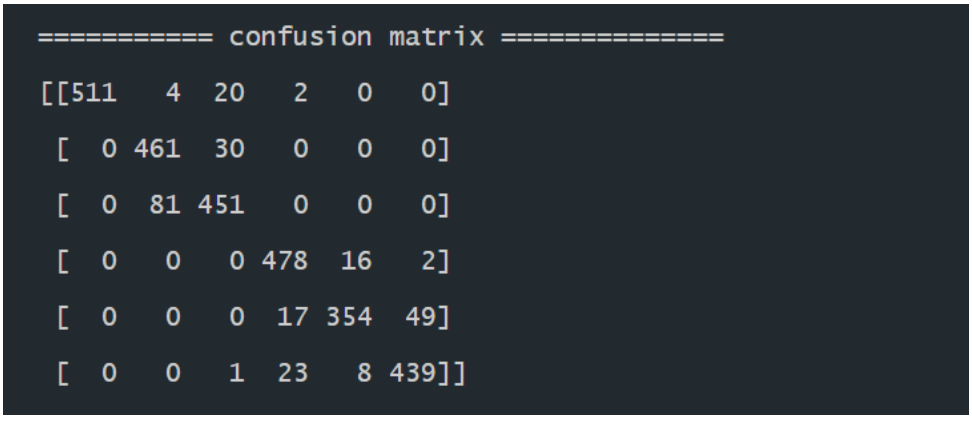
In the function “k_cross_validation(k)”, the thing you need to be aware is that, before each epoch of cross validation, we should first restore the initial parameter of the model, because each epoch of cross-validation must base on a same parameter state (must have the same initial parameter), otherwise, different epoch of CV performance will has no comparability.

After 50-fold CV, we found the average CV performance (testing loss) is 1.79, which is relatively high.

```
50-fold round # 38, loss = 1.79
50-fold round # 39, loss = 1.79
50-fold round # 40, loss = 1.79
50-fold round # 41, loss = 1.79
50-fold round # 42, loss = 1.79
50-fold round # 43, loss = 1.79
50-fold round # 44, loss = 1.79
50-fold round # 45, loss = 1.79
50-fold round # 46, loss = 1.79
50-fold round # 47, loss = 1.79
50-fold round # 48, loss = 1.79
50-fold round # 49, loss = 1.79
50-fold round # 50, loss = 1.79
50-fold CV's average loss = 1.79
```

(Figure-14)

```
========== confusion matrix ==============
[[511   4  20   2   0   0]
 [  0 461  30   0   0   0]
 [  0  81 451   0   0   0]
 [  0   0   0 478  16   2]
 [  0   0   0  17 354  49]
 [  0   0   1  23   8 439]]
```



(Figure-15)

In Figure-15, you can check the confusion-matrix, The interpretation is like the following Figure-16.

For the class “Laying”, there are 528 actual Layings, 528 was correctly classified and 0 was wrongly classified, meaning the classification performance on Laying is very good, accuracy = 100%. However, for class Sitting, 435 were correctly classified but 56 were wrongly classified, accuracy = 88.6%, relatively low.

Actual \ Predict	Laying	Sitting	Standing	Walking	Walking_Downstairs	Walking_Upstairs	
Laying	528	0	9	0	0	0	537
Sitting	1	435	52	0	0	3	491
Standing	0	40	491	0	0	1	532
Walking	0	0	0	487	9	0	496
Walking_Downstairs	0	0	0	26	328	66	420
Walking_Upstairs	0	0	1	2	0	468	471

(Figure-16)

The average metrics is around 0.91 an 0.92, not bad.

```
========== classification report ==============
                    precision    recall  f1-score   support

           LAYING       1.00      0.95      0.98       537
          SITTING       0.84      0.94      0.89       491
         STANDING       0.90      0.85      0.87       532
          WALKING       0.92      0.96      0.94       496
WALKING_DOWNSTAIRS       0.94      0.84      0.89       420
   WALKING_UPSTAIRS       0.90      0.93      0.91       471

         accuracy                           0.91      2947
        macro avg       0.92      0.91      0.91      2947
     weighted avg       0.92      0.91      0.91      2947
```

(Figure-16)

2. Run the SVM training and evaluation process:

```
Best score for training data:0.9876221923167634
Best C: 100
Best Kernel: rbf
Best Gamma: 0.001
```

The above {kernel = RBF, C=100, Gamma = 0.001} is the best parameter selection for the SVM model. Under such parameters, the performance of SVM can reach an average of 0.96 on {precision, accuracy, recall, f1}, which is very high.

```
[[537   0   0   0   0   0]
 [  4 449  37   0   0   1]
 [  0  19 513   0   0   0]
 [  0   0   0 477  10   9]
 [  0   0   0   4 396  20]
 [  0   0   0   4   2 465]]
```

```
                    precision    recall  f1-score   support

           LAYING       0.99      1.00      1.00       537
          SITTING       0.96      0.91      0.94       491
         STANDING       0.93      0.96      0.95       532
          WALKING       0.98      0.96      0.97       496
WALKING_DOWNSTAIRS       0.97      0.94      0.96       420
  WALKING_UPSTAIRS       0.94      0.99      0.96       471

         accuracy                           0.96      2947
        macro avg       0.96      0.96      0.96      2947
     weighted avg       0.96      0.96      0.96      2947

Training set score for SVM: 1.000000
Testing  set score for SVM: 0.962674
```

3. Run the Random Forest training and evaluation process:

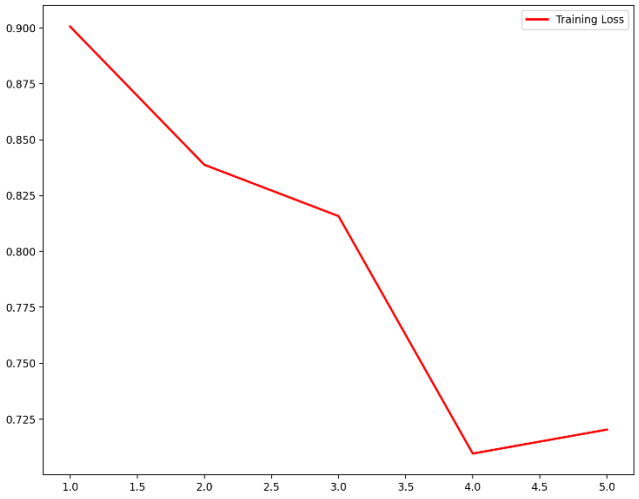
```
分类的正确率为：0.9114353579911775
[[537   0   0   0   0   0]
 [  0 481  10   0   0   0]
 [  0 155 377   0   0   0]
 [  0   0   0 470  19   7]
 [  0   0   0   7 379  34]
 [  0   0   0  19  10 442]]
```

	precision	recall	f1-score	support
LAYING	1.00	1.00	1.00	537
SITTING	0.76	0.98	0.85	491
STANDING	0.97	0.71	0.82	532
WALKING	0.95	0.95	0.95	496
WALKING_DOWNSTAIRS	0.93	0.90	0.92	420
WALKING_UPSTAIRS	0.92	0.94	0.93	471
accuracy			0.91	2947
macro avg	0.92	0.91	0.91	2947
weighted avg	0.92	0.91	0.91	2947

We can see the random forest reached an average score around 0.91 and 0.92 on { precision, accuracy, recall, f1}, which is lower than SVM but the same as Neural Network, not bad.

4. Run the Bert training and evaluation process:

```
Epoch 1, Loss: 0.9006242156028748
Epoch 2, Loss: 0.8387940526008606
Epoch 3, Loss: 0.8158231973648071
Epoch 4, Loss: 0.7096035480499268
Epoch 5, Loss: 0.7203267812728882
```



For the Bert model, you can download the model file in HuggingFace to your local computer from this link: https://huggingface.co/google-bert/bert-base-uncased/tree/main

For this model, we did not get any performance data, because each epoch of training evaluation costs more than 1 hour, which is too long. Since we are using the Nvidia Quadro P4000 GPU to test this model with only 8GB RAM, the training speed is very low, so I suggest you to use at least RTX3070 to test the Bert.

The core idea of running Bert is that:

1. When handling the dataset, we combine each sample's 561 float features into one single long string text, which each float separated by a comma.
2. The reason of combing 561 features into a string is that, the Bert model can only handle text classification problem.
3. We let the Bert to classify the long string text into 1 of the 6 classes.
4. Because the Bert is not a multi-classifier, we assuming its performance is very bad and around 0.6. (Again, we did not get the real performance data)
5. Bert's training and testing dataset must be handled in a special format:

```python
class NewsDataset(Dataset):
    def __init__(self, texts, labels, tokenizer):
        self.texts = texts
        self.labels = labels
        # 将文本转化为模型能够理解的格式，  比如tokens
        self.tokenizer = tokenizer
        self.encoder = LabelEncoder().fit(self.labels)
        self.labels = self.encoder.transform(self.labels)

    def __len__(self):
        '''
        return the length of the text
        '''
        return len(self.texts)

    def __getitem__(self, idx):
        '''
        :idx: 索引

        :return
        '''
        text = self.texts[idx]
        label = self.labels[idx]

        # 将文本分割为令牌
        # 添加特殊的开始和结束令牌
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            # 不足64，则自动填充
            max_length=64,
            return_token_type_ids=False,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt',
            truncation=True
        )
```

```python
        '''
        'input_ids': 这是文本经过分词器处理后的结果，
                        每个词都被转换成了一个唯一的ID。这些ID是模型的输入，模型会用它们来查找词嵌入。

        'attention_mask': 这是一个与输入ID (input_ids) 相同长度的二进制向量，
                        用于指示哪些词是实际的词，哪些词是填充词。
                        例如，如果输入序列的长度小于64，那么剩余的位置会被填充为0，
                        对应的attention mask也会被设置为0。模型会忽略mask为0的词。

        'labels': 这是文本的标签，也就是我们希望模型预测的目标。
        '''


        return {
            # flatten: transfer multi-dimension vector into one-dimension vector

                    # 将input_ids张量从形状(batch_size, sequence_length)
                    # 转换为形状(batch_size * sequence_length,)

            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label)
        }
```

We can see that, in the above code, each input string text is tokenized into many tokens and each token corresponds to a unique input id, all those ids are stored in encoding[“input_ids”].

```python
def main()-> nn.Module:

    # 加载数据集
    X_train_combine, Y_train, Y_train_label, X_test_combine, Y_test, Y_test_label, le = build_dataset()


    # 初始化Tokenizer和模型
    # 加载了预训练的BERT模型和分词器。num_labels=6指定了模型的输出类别数。
    # tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
    tokenizer = BertTokenizer.from_pretrained('D:/huggingFace/bert_model')

    model = BertForSequenceClassification.from_pretrained('D:/huggingFace/bert_model', num_labels=6)




    # train_texts, test_texts, train_labels, test_labels = train_test_split(texts, labels, test_size=0.2)
    train_dataset = NewsDataset(X_train_combine, Y_train_label, tokenizer)
    test_dataset = NewsDataset(X_test_combine, Y_test_label, tokenizer)
    # shuffle means randomly pick 100 samples as a batch
    train_loader = DataLoader(train_dataset, batch_size=100, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=100)

    # train_loader['input_ids']

    # 训练模型

    # create a "device" object
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # 将模型的所有参数和缓冲区移动到指定的设备上
    model = model.to(device)
```

```python
    # why use AdamW:
    # 1. AdamW is a variation of Adam, it uses L2 regularization to utilize "weight decay" to prevent overfitting
    # 2. Bert has large quantity of parameters and very easy to overfitting
    optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)
    watch_loss = []
    for epoch in range(3):  # 训练3个周期
        model.train()
        for batch in train_loader:
            # 创建了一个字典推导式， k是键， v是k对应的张量， v.to(device)将张量移动到设备上
            # batch字典可能包含了如input_ids，attention_mask等模型需要的输入数据
            batch = {k: v.to(device) for k, v in batch.items()}
            # **操作符是Python中的解包（unpacking）操作符，
            # 它会将字典batch中的键值对作为关键字参数传递给model函数。
            outputs = model(**batch)
            loss = outputs.loss
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        watch_loss.append(loss.item())
        print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

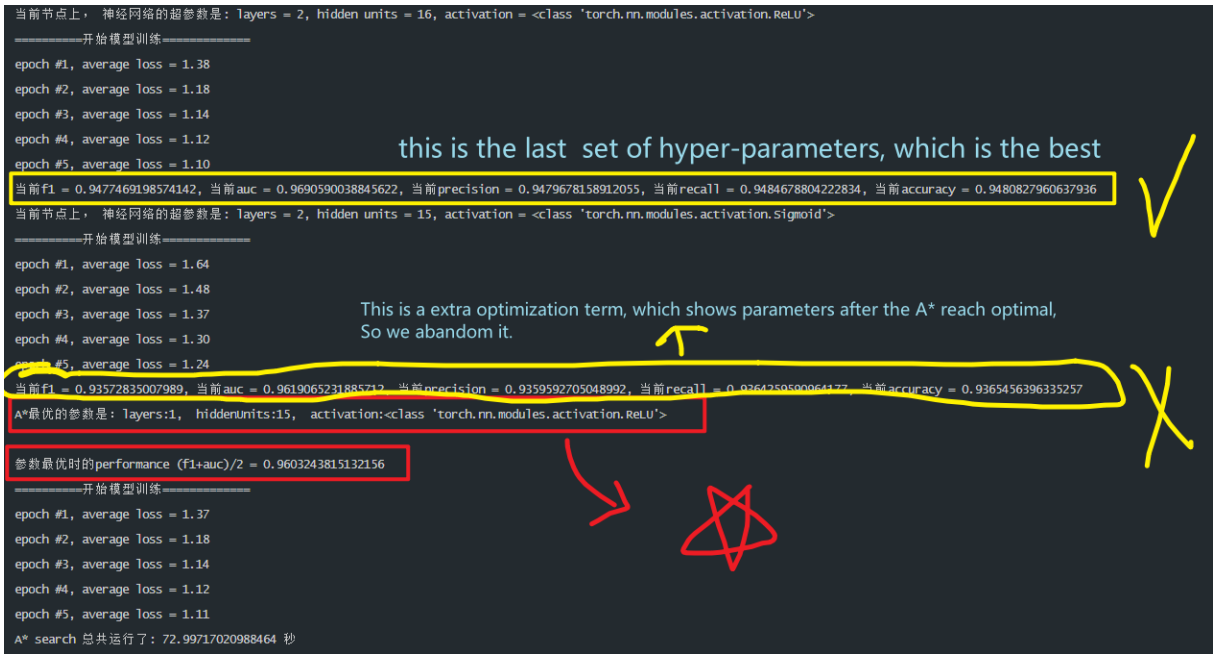
For the Bert model's training part, we first import the model and the tokenizer, and then we detect whether the cuda exists, if it exists, we transfer the model and the dataset to the cuda, accelerating the training.

Just like the neural network, we calculate the gradient, using BP to update the weight matrix and then clear the gradient, and doing all of those using AdamW optimizer, which it use L2 regularization to add to the loss function and prevent overfitting.

5. Run the A* optimized Neural Network's training and evaluation process:

```
========== A* 预计要运行 10分钟 ========
当前节点上， 神经网络的超参数是: layers = 4, hidden units = 10, activation = <class 'torch.nn.modules.activation.ReLU'>
=========开始模型训练==========
epoch #1, average loss = 1.58
epoch #2, average loss = 1.31
epoch #3, average loss = 1.19
epoch #4, average loss = 1.14
epoch #5, average loss = 1.12
当前f1 = 0.9245375747446447, 当前auc = 0.9555466329928337, 当前precision = 0.9259061083148902, 当前recall = 0.9259070008275389, 当前accuracy = 0.9253478113335596
当前节点上， 神经网络的超参数是: layers = 6, hidden units = 10, activation = <class 'torch.nn.modules.activation.ReLU'>
=========开始模型训练==========
epoch #1, average loss = 1.67
epoch #2, average loss = 1.31
epoch #3, average loss = 1.20
epoch #4, average loss = 1.15
epoch #5, average loss = 1.13
当前f1 = 0.9162595670094212, 当前auc = 0.9503058275517313, 当前precision = 0.9172976313849611, 当前recall = 0.9169012589842943, 当前accuracy = 0.9178825924669155
当前节点上， 神经网络的超参数是: layers = 5, hidden units = 9, activation = <class 'torch.nn.modules.activation.ReLU'>
=========开始模型训练==========
epoch #1, average loss = 1.61
epoch #2, average loss = 1.45
epoch #3, average loss = 1.41
epoch #4, average loss = 1.38
epoch #5, average loss = 1.37
当前f1 = 0.5426675254667831, 当前auc = 0.7988465918633261, 当前precision = 0.46218512421325514, 当前recall = 0.6611476382896149, 当前accuracy = 0.6854428232100441
```

```
当前f1 = 0.9095870731180886, 当前auc = 0.9469733493369108, 当前precision = 0.913168692876528, 当前recall = 0.9113498264210205, 当前accuracy = 0.9121140142517815
当前节点上， 神经网络的超参数是: layers = 5, hidden units = 10, activation = <class 'torch.nn.modules.activation.Sigmoid'>
=========开始模型训练==========
epoch #1, average loss = 1.79
epoch #2, average loss = 1.78
epoch #3, average loss = 1.77
epoch #4, average loss = 1.75
epoch #5, average loss = 1.71
当前f1 = 0.17315730971405108, 当前auc = 0.6010000434519345, 当前precision = 0.11696671244837235, 当前recall = 0.3333333333333333, 当前accuracy = 0.3505259586019681
当前节点上， 神经网络的超参数是: layers = 3, hidden units = 10, activation = <class 'torch.nn.modules.activation.ReLU'>
=========开始模型训练==========
epoch #1, average loss = 1.55
epoch #2, average loss = 1.24
epoch #3, average loss = 1.16
epoch #4, average loss = 1.13
epoch #5, average loss = 1.12
当前f1 = 0.9415503764104701, 当前auc = 0.9653804972974384, 当前precision = 0.9416620866701785, 当前recall = 0.9422412794279288, 当前accuracy = 0.9423142178486597
当前节点上， 神经网络的超参数是: layers = 5, hidden units = 10, activation = <class 'torch.nn.modules.activation.ReLU'>
=========开始模型训练==========
epoch #1, average loss = 1.67
epoch #2, average loss = 1.45
epoch #3, average loss = 1.35
epoch #4, average loss = 1.27
epoch #5, average loss = 1.24
当前f1 = 0.8236293032853054, 当前auc = 0.8946920192453316, 当前precision = 0.8273096720448622, 当前recall = 0.8254118440651655, 当前accuracy = 0.8201560909399389
当前节点上， 神经网络的超参数是: layers = 4, hidden units = 9, activation = <class 'torch.nn.modules.activation.ReLU'>
```



You can see that, after 73 seconds of training, the best neural network parameters are {layers =1, hidden units = 15, activation = ReLU}, such combination is not always the same among different rounds of testing. Also we can see that the best performance reached a 0.96, where the NN without A* is 0.92, and SVM is 0.9695.

We can say that, the A* optimized NN is better than the previous NN, but still can not reach the performance of SVM, however, the distance between SVM and A* optimized NN has become smaller.

Discussion

SVM's classification performance can surpass all other models (Maybe not include Bert since we do not have performance data for Bert).

Regular Neural Network's performance is nearly the same as the random forest, and there is a gap between NN and SVM, about 0.04.

A* algorithm can be used to find better hyper-parameters like layers and hidden units that can maximize the performance of the NN.

However, whether A* can find the best parameters depends on the setting of the goal, if the goal is too large, like 0.97, then A* search may never find that parameter combination where the model performance can excel 0.97, or maybe it can find eventually but will take hours to run the algorithm.

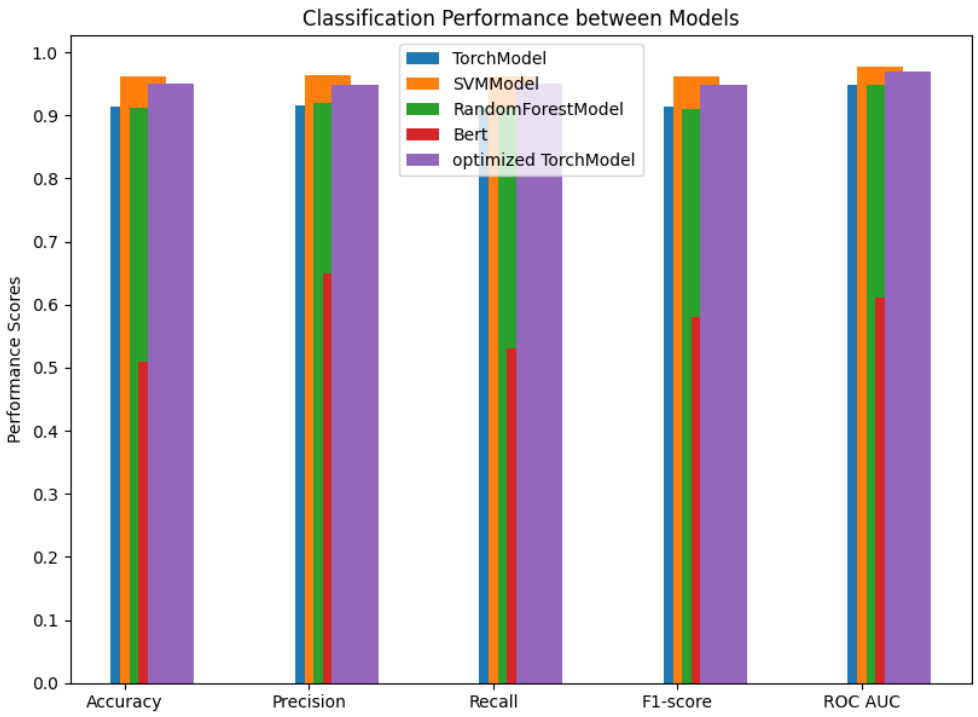
In our case, we set the A*'s goal to be 0.96, which is slightly lower than SVM's performance but it is a easy goal for A* to reach.

The consequence is that, under the goal of 0.96, A* optimized NN can only approach but never excel the SVM (0.9675), this is a trade off for the A* search time. However, even in this case, the A* optimized NN still has a huge improvement compare

to the old version of NN (improve by 0.04), which shows a great potential of combining parameter heuristic searching to the Neural Network.

For the SVM model, we have noticed that the model training time is too long (2 minutes), I think this is due to the time of Grid Search where it must compare different combination of parameters in the grid and calculate many performance scores, which can take a long time.

Compare 5 models' performance metrics:




```
Accuracy        Precision       Recall  F1      AUC
model-0: [0.9331523583305056, 0.9354553261540386, 0.9326552659381768, 0.9325773376430804, 0.959650456333052]
model-1: [0.9626739056667798, 0.9630370993523759, 0.9617596395265254, 0.9621095358584367, 0.9771426678649161]
model-2: [0.9253478113335596, 0.9284364359766094, 0.9254475958664942, 0.9243163972101422, 0.9552908780340027]
model-3: [0.51, 0.65, 0.53, 0.58, 0.61]
model-4: [0.9460468272819816, 0.94534084221385, 0.9459121948927592, 0.9455012471094592, 0.9675749577615561]
```

We can see from the comparison result that, SVM (model-1) reach a best performance, each metric of SVM is higher then 0.96, the highest is AUC score that reaches a 0.977. We did set the A* goal to be AUC=0.977, but the experiment shows A* will never find the best parameters for NN under such goal. Later, we even set the A*'s goal to be $\frac{accuracy + precision + recall + f1 + auc}{5} = 0.9695$, but the experiment also shows that A* can never find such parameters for NN, meaning under current experiment set ups, A* optimized NN (model-4) can never excel SVM.

In the regular Neural Network (model-0), we found its average performance is higher than random forest (model-2) (0.9386 vs. 0.9314), meaning the fundamental

performance of NN is still excellent.

For A* optimized Neural Network (model-4), although it can not compete with SVM, it can still beat other models (0.95 vs. (NN = 0.9386, RF=0.99314)), we can still say A* is efficient on enhancing NN model's performance.

Conclusion

SVM is the best multi-classification model over all 5 metrics (accuracy, precision, recall, F1, auc), it reaches a highest average score of 0.9654. So, we can accept H2. The only shortcoming of the SVM is that its training time is too long, we predict that this is caused by the Grid Search algorithm, we think it is a brute-force search algorithm since it must traverse every combination of parameters. If the parameter list gets longer, I suppose the training time will also become longer, so the usage of heuristic search in SVM should be taken into consideration. For example, A*, UCS, Greedy, Ant Colony and Simulated Annealing.

The performance of the Neural Network without A* is nearly the same as the random forest, but slightly better.

After implementing A* to search for the best parameters for Neural network, its average performance increases from 0.9386 to 0.95, meaning A* is efficient in enhancing NN's performance. However, it is still hard for NN to excel SVM, maybe it requires others parameter searching skills (Mountain Climbing, Tabu, Simulated Annealing). Or, we can implement other not-heuristic search method like normal quick search, or AVL, or Red-Black Tree.

Under such observation, we must reject H3, because in current stage, even with A*, NN can never excel SVM, no matter which performance metric they are comparing.

As for H1, we do transfer the multi-classification task into a text-classification task, because we did successfully run the model training and get the loss history. However, we can not say that Bert reaches an average performance of other models since we do not have enough evidence.

References

1. Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436–444. https://doi.org/10.1038/nature14539

2. Cherkassky, V., & Ma, Y. (2004). Practical selection of SVM parameters and noise estimation for SVM regression. Neural Networks, 17(1), 113–126. https://doi.org/10.1016/S0893-6080(03)00169-2

3. Zaremba, W., Sutskever, I., & Vinyals, O. (2015). Recurrent Neural Network

Regularization (arXiv:1409.2329). arXiv. http://arxiv.org/abs/1409.2329

4. Cortes, C., & Vapnik, V. (1995). Support-vector networks. Machine Learning, 20(3), 273–297. https://doi.org/10.1007/BF00994018

5. Patle, A., & Chouhan, D. S. (2013). SVM kernel functions for classification. 2013 International Conference on Advances in Technology and Engineering (ICATE), 1–9. https://doi.org/10.1109/ICAdTE.2013.6524743