موسسه آموزش عالی آزاد توسعه

برگزار کننده دوره‌های تخصصی علم داده

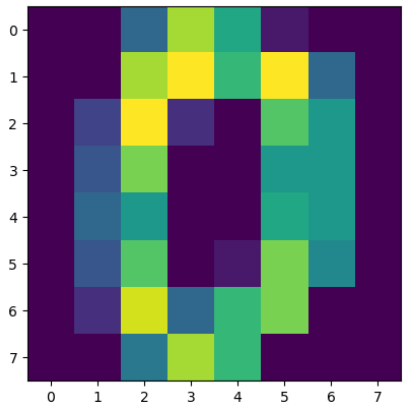
Homework 8: Fall 2020
Due date: (۱۶ آبان)
Please email your HWs to y.zerehsaz@gmail.com

*****Please hand in your HWs as a word file with your email as the document's name. For instance, I would name my word file as y.zerehsaz@gmail.com.docx.**
*****Make sure to copy and paste the codes that you used for each question. I need to see your plots, results and conclusions but not the long output of your codes.**
*****When asked, please explain your results.**

For this homework, you need to use the digits dataset in Python located in sklearn package.

```
from sklearn.datasets import load_digits
digits = load_digits(return_X_y =True)
X=digits[0]
y=digits[1]
```

Each row in X is vectorized image of a number between 0 and 9. The images can be visualized by this code: plt.imshow(X[0,:].reshape(8,8))



What we would like to do in this homework is to use several methods to perform a digit classification. Digits classification is the act of labeling every image with its corresponding digit. The response variable; as a result, is a number between 0 and 9 given by y.

We need to apply SVM (linear, RBF and polynomial), decision tree, bagging, boosting, random forest and convolutional neural network to the data.

1- Perform a crossvalidation to obtain:

a) The tuning parameter c in linear SVM.

b) Tuning parameter c, gamma and d in both RBF and polynomial SVM. (RBF does not have d)

Clarifications and hints:

Normally for crossvalidaton, we can write a for loop and use cross_val_ score function to crossvalidate the tuning parameter c or the γ (gamma) parameter in Radial Basis Function (RBF) kernel. It is totally OK, but we have another way in Python to do crossvalidation, which some people might find easier. The GridSearchCV function in model_selection folder inside sklearn package performs a wide range of crossvalidations on the hyperparameters of almost all methods which we have learnt so far.

In SVM, we have several hyperparameters to decide about. First, the type of kernel. Whether it is linear, RBF or Polynomial. Depending on the selected kernel, we might have one or two additional hyperparameters. For instance, if we choose RBF kernel, we can crossvalidate c and γ. What I am going to do next is to explain GridSearchCV function and its arguments, then I will perform a crossvalidation on the kernels (linear or RBF) and their associated hyperparameters c and γ.

What will be left for you to do in this question?

Well, you will need to do a crossvalidation for the linear, RBF and Polynomial kernels altogether. Note that for the Polynomial kernel, you need to crossvalidate c, γ and the degree of the polynomial d. Recall that the polynomial kernel in Python is given by

$$K(\mathbf{x}^*, \mathbf{x}) = (\gamma\langle \mathbf{x}^*, \mathbf{x}\rangle + 1)^d$$

Let's take a look at the GridSearchCV function. The function is defined as

Class sklearn.model_selection.GridSearchCV(*estimator, param_grid, scoring, cv=None*)

And the arguments are

estimator: is the estimator that you would like to use in your problem

param_grid: it is a dictionary of the hyperparameters along with their ranges that must be used for crossvalidation

cv: number of folds in crossvalidation

scoring: the type of score function you would like to use for crossvalidation (accuracy, precision, recall, …).

Now, we can do some crossvalidation, but you need to call the required libraries and split train and test data first

```
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2)
```

crossvalidate the hyperparameter c in a linear SVM

The first argument in GridSearchCV is the *estimator*. To define the estimator, just define an SVM model without selecting the hyperparameters:

```
svm_clf = svm.SVC()
```

Second, you need to define the hyperparameters. Typically, a dictionary is used to define all parameters as

```
Hyperparameters= {'C':np.arange(0.01,11,1),'kernel':['linear']}
```

The variable Hyperparameters is a dictionary with two keys. The first key is the tuning parameter c which should be written in a capital-letter style 'C' since this is the acceptable format in SVM. After defining the first key, you can define a range of values to try for your tuning parameter. I used a list of numbers including 0.01, 1.01, 2.01, …, 10.01. Every time, Python sets c to one of these values and performs a cv-fold crossvalidation. A score function defined in *scoring* argument is computed, and the average is reported. We select a value optimizing the score function.

The second key is the 'kernel' which we would like to employ in SVM. In front of this key, we can define a list of values, like 'linear', 'rbf' and 'poly'. I will just define 'linear' here. Now, our hyperparameters are ready for crossvalidation.

```
cv_grid = GridSearchCV(estimator= svm_clf ,param_grid = hyperparameters, scoring ='accuracy',cv = 10)
cv_grid= cv_grid.fit(X_train, y_train)
```

Note that I used 'accuracy' in the *scoring* argument. A 10-fold crossvalidation will be used, and the parameter maximizing the validation accuracy will be selected.

```
cv_grid.best_params_# this code gives you the best hyperparameters
{'C': 1.01, 'kernel': 'linear'}
```

So, the best result is obtained at $c = 1.01$. How much is the crossvalidation accuracy?

Your results might be different.

```
cv_grid.best_score_# this code gives the best crossvalidation score (it is accuracy here)
0.9763500388500388
```

Can we get the crossvalidation score for the whole range of values defined in the variable Hyperparameters? Yes, let's see how it is done

```
cv_grid.cv_results_['params']# this code gives you all the parameters used in your crossvalidation
[{'C': 0.01, 'kernel': 'linear'},
 {'C': 1.01, 'kernel': 'linear'},
 {'C': 2.01, 'kernel': 'linear'},
 {'C': 3.01, 'kernel': 'linear'},
 {'C': 4.01, 'kernel': 'linear'},
 {'C': 5.01, 'kernel': 'linear'},
 {'C': 6.01, 'kernel': 'linear'},
 {'C': 7.01, 'kernel': 'linear'},
 {'C': 8.01, 'kernel': 'linear'},
 {'C': 9.01, 'kernel': 'linear'},
 {'C': 10.01, 'kernel': 'linear'}]
```

And the following code gives you the crossvalidation score computed for each value in the given range

```
cv_grid.cv_results_['mean_test_score']

array([0.97773893, 0.97843337, 0.97843337, 0.97843337, 0.97843337,0.97843337,
0.97843337, 0.97843337, 0.97843337, 0.97843337,0.97843337])
```

Can we get the best classifier and apply it on the test data? Yes, here is how:

```
cv_grid.best_estimator_.fit(X_train, y_train)#get the best estimator and fit it on the train data
cv_grid.score(X_test,y_test)# compute the testing accuracy
y_pred=cv_grid.predict(X_test)
confusion_matrix(y_test,y_pred)
array([[47,  0,  0,  0,  1,  0,  0,  0,  0,  0],
       [ 0, 31,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 37,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0, 41,  0,  0,  0,  0,  1,  0],
       [ 0,  0,  0,  0, 28,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0, 38,  0,  0,  0,  1],
       [ 0,  0,  0,  0,  1,  0, 26,  0,  0,  0],
       [ 0,  0,  0,  0,  1,  0,  0, 38,  0,  0],
       [ 0,  2,  0,  0,  0,  0,  0,  0, 36,  0],
       [ 0,  0,  0,  0,  0,  1,  0,  0,  1, 29]], dtype=int64)
```

Crossvalidate the hyperparameters c, γ and the kernels in SVM

In this section, I will use the GridSearchCV function to crossvalidate c, γ and kernels. First, we need to define a dictionary of hyperparameters and their corresponding ranges.

```
Hyperparameters = {'C':np.arange(0.01,11,1),'gamma':np.arange(0.001,0.01,0.001),'kernel':['linear','rbf']}
```

Next, we will perform a grid search on the hyperparameters' space to find the best setting.

```
svm_clf = svm.SVC()
cv_grid= GridSearchCV(estimator= svm_clf ,param_grid = Hyperparameters, scoring ='accuracy',cv = 10)
cv_grid= cv_grid.fit(X_train, y_train)
cv_grid.best_params_
```

```
{'C': 2.01, 'gamma': 0.001, 'kernel': 'rbf'} # this is the best setting for hyperparameters, and we can use it for prediction
```

```
cv_grid.best_estimator_.fit(X_train, y_train)
cv_grid.score(X_test,y_test)
0.986
```

Your Task for part b of Question 1:

Now, it is your turn to work on this question. Please crossvalidate the hyperparameters c, γ, degree of polynomial d and three kernels (linear, rbf and poly) and find the best setting.

c) The parameter ccp_alpha in decision tree. If you do not have sklearn version 0.22, you can cross validate max_depth parameter.

d) Number of estimators for bagging. You can use the best tree obtained from part c as your base estimator.

e) Number of estimators (M in slides) for boosting. You can use the best tree obtained from part c as your base estimator.

f) Number of estimators and max_depth in random forest.

2- Reshape the images into 8×8 arrays and use CNN to classify your dataset. Compare the testing accuracy of all methods and introduce the best. You need to decide on the number of filters, filter size, whether to dropout or not, whether to pool layers or not, number of epochs, batch size, etc.

This is how you can reshape:
X_train=X_train.reshape((X_train.shape[0],8,8,1))
X_test=X_test.reshape((X_test.shape[0],8,8,1))

Clarifications and hints:

Can we apply a rigorous grid search strategy to find the networks' parameters like what we did in Question 1? Off course yes, but the there is a problem here!

What is the problem? grid search is not efficient for the neural networks. Why? Because there are so many parameters in NN, and it will take forever to crossvalidate all these parameters.

What is the solution then? We need to randomly select some of the parameters from the grid and perform the cross validation. Let me give you an example. Suppose that you are using a CNN, and you need to decide on the number of filters in your layers, filter size, stride, the number of epochs and batch size. Here are some suggestions:

N_filters=[16, 32, 64, 128, 256, 400]
Filter_size=[2, 3, 5, 7]
Strides=[1, 2, 3]
N_epochs=[5, 10, 20, 50, 70]
Batch_size=[20, 50, 150, 350]

How many different combinations are we going to have? $6 \times 4 \times 3 \times 5 \times 4 = 1440$ combinations! For each of these combinations, a 10-fold crossvalidation must be used, and the best setting should be reported. It is, indeed, time consuming. Random selection is a way to select only some combinations out of 1440 possible combinations that we have.
Does randomized search loose quality against grid search (GridSearchCV)? off course it does, and you are welcome to try GridSearchCV if you have enough time, but this might take days to run.

So, let's talk about this random search. Python has a function called RandomizedSearchCV which is very similar to GridSearchCV, and it is located in sklearn.model_selection. Here is the class

class `sklearn.model_selection.`**`RandomizedSearchCV`**(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *cv=None*)

All arguments are similar to those of GridSearchCV except for *n_iter* which is the number of combinations considered by the random search. As a result, we can control the number of combinations used for crossvalidation. This property enables us to have a more timely -effective search over the parameter space, which is specifically useful in training a neural network.

Having discussed random search, I must add that preparing a neural network for crossvalidation is not easy. There are different actions that can be taken. For instance, we can decide on the number of layers first and then perform the crossvalidation on the rest of hyperparameters. So, for example, we can say that we need a network with two convolutional layers and one dense layer and perform a crossvalidation to find the best number of filters, filter size, stride, number of epochs, etc. This is exactly what you are going to do in this Question.

We need to build a network with three hidden layers consisting of two convolutional layers followed by a pooling layer and a fully connected dense layer at the end. Here is a list of all hyperparameters that must be determined:

- number of filters in both layers
- filter size in both layers
- strides in both layers
- the number of units in the dense layer
- the number of epochs
- batch size
- learning rate in the ADAM algorithm
- Whether to drop out or not

What we normally do to crossvalidate these parameters is to find a range for each parameter, define a dictionary called "Hyperparameters", and give it to RandomizedSearchCV function. What would be the first argument of this function? we know that our first argument is the estimator. In SVM, the first argument is svm.SVC(). When choosing this argument, RandomizedSearchCV extracts the parameters from the Hyperparameters argument and plugs them in svm.SVC(). The model will be fitted, afterwards, and the crossvalidation will be done. What happens in neural network? to understand this, we need to review the different steps leading to building a network.

Step 1: Use the keras.models.Sequential() function to add the layers. This is how it will look like

```
model=keras.models.Sequential()
model.add(keras.layers.Conv2D(filters=50, kernel_size=5, strides=1, activation='relu', input_shape=(8,8,1),
padding='same'))
model.add(keras.layers.Conv2D(filters=150, kernel_size=3, strides=1, activation='relu'))
model.add(keras.layers.MaxPool2D(pool_size=2, strides=2))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(160, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

Step 2: compile the model

```
model.compile(optimizer=tf.compat.v1.train.AdamOptimizer(learning_rate=0.01),
loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

Step 3: fit the model

```
model.fit(X_train, y_train, epochs = 10, batch_size = 50)
```

Which model should be given to RandomizedSearchCV as the estimator? Off course we cannot try the model before it is compiled. The model should be definitely compiled first. So, we need to define the layers in Step 1 and compile the model in Step 2.

Should we give the model before fitting it at Step 3? Yes, this is what we did in all previous methods. Well then, we would compile the model and plug it in RandomizedSearchCV. However, we have several parameters in Step 1 that must be defined so that we can complete the step. For example, the number of filters or filter size needs to be known; on the other hand, these are the hyperparameters that must be determined!

See, this is the problem here in neural network. In SVM, we did not have such a problem because we did not need Step 1 and Step 2. We could just ask RandomizedSearchCV to take svm.SVC() along with the hyperparameters containing c, gamma, d and the kernels. It would plug these parameters in svm.SVC(), fit the model, and that was it! But here, the first two steps must be completed first, and they cannot be completed without the hyperparameters being known, which is a contradiction!

What is the solution? Functions to the rescue! Use a function to define steps one and two. Let's do this.

```
def bl_model(nf1,nf2,ks1,ks2,n_units,lrt):
        model=keras.models.Sequential()
        model.add(keras.layers.Conv2D(filters=nf1, kernel_size=ks1, strides=1, activation='elu',
        input_shape=(8,8,1), padding='same'))
        model.add(keras.layers.Conv2D(filters=nf2, kernel_size=ks2, strides=1, activation='relu'))
        model.add(keras.layers.MaxPool2D(pool_size=2, strides=2))
        model.add(keras.layers.Flatten())
        model.add(keras.layers.Dense(n_units, activation='relu'))
        model.add(keras.layers.Dense(10, activation='softmax'))
        model.compile(optimizer=tf.compat.v1.train.AdamOptimizer(learning_rate=lrt),
        loss='sparse_categorical_crossentropy', metrics=['accuracy'])
        return(model)
```

A function is defined to construct two convolutional layers along with a pooling layer and a dense layer at the end. The hyperparameters are the inputs; consequently, they do not need to be specified. The last line compiles the model, and the learning rate is considered as a hyperparameter. The output of this model can be given to RandomizedSearchCV function now. We can take a look at a sample output of this function in Table 1.

```
Model = bl_model(nf1 = 50, nf2 = 150 ,ks1 = 3, ks2 = 3 , n_units = 100, lrt = 0.01)
Model.summary() # Get a summary of your model
```

Table 1. Summary of the simple CNN model

Layer (type)	Output Shape	Param #
conv2d_1232 (Conv2D)	(None, 8, 8, 50)	500
conv2d_1233 (Conv2D)	(None, 6, 6, 150)	67650
max_pooling2d_561 (MaxPooling)	(None, 3, 3, 150)	0

```
_________________________________________________________________
flatten_695 (Flatten)        (None, 1350)              0
_________________________________________________________________
dense_1367 (Dense)           (None, 100)               135100
_________________________________________________________________
dense_1368 (Dense)           (None, 10)                1010
=================================================================
Total params: 204,260
Trainable params: 204,260
Non-trainable params: 0
```

The summary of your model is given in Table 1. The type of layers, shape of the output layer and the number of parameters that must be estimated in each layer are provided. If you are curious about understanding the summary, please see appendix A.

So far, we have learnt that the function bl_model can be delivered to RandomizedSearchCV, but the hyperparameters should be defined first. We can use a dictionary to define the hyperparameters along with their corresponding ranges.

```
params={'nf1':[32],'nf2':[256],'ks1':[3],'ks2':[3],'n_units':[128],'lrt':[0.001],'epochs':[10],'batch_size':[30]}
```

It seems that everything is ready to go except for one small problem. RandomizedSearchCV function does not accept an estimator defined outside sklearn! What can be done? Easy, we can apply a scikit-learn wrapper on our model and change it to sklearn model. Here is how it goes

```
from keras.wrappers.scikit_learn import KerasClassifier

final_model=KerasClassifier(bl_model)
```

Now, we are really ready to do some crossvalidations.

```
from sklearn.model_selection import RandomizedSearchCV

cnn=RandomizedSearchCV(final_model,param_distributions=params, n_iter=10, scoring='accuracy',  cv=5)

cnn.fit(X_train,y_train)
```

What is left for you to do?

First, use wider sets of ranges for all the hyperparameters defined in 'params' above. For instance, you can use a list of [16, 32, 64, 128] for the number of filters in your first layer; i.e. 'nf1': [16, 32, 64, 128]. You should select wider ranges for all your parameters. Run the function again and fit the model. Get the best set of parameters. You can use a five-fold crossvalidation to reduce the running time.

Second, you need to modify bl_model function to decide on doing drop out or not. This means that you have to add an input to the function and change the function body. The input can be in the format of true/false which if true, a drop out layer must be added.

Add the layer right before your dense layer. The probability of discarding a unit can be set to 0.5. Here is how the layer should look like

```
model.add(keras.layers.Dropout(0.5))
```

3- Please apply the best SVM, decision tree, bagging, boosting, random forest and the best CNN that you found in Questions 1 & 2 to the data. Repeat the whole procedure 1000 times. Compute the mean and standard deviation of accuracy. You need to fill in the gaps in the following table using the test data.

	Accuracy Mean (standard deviation)
SVM	
Bagging	
Boosting	
Decision Tree	
Random Forrest	
CNN	

Good luck!

Appendix A:

In this appendix, I like to take a deeper look at the summary of the model presented in Table 1. I will, first, discuss the output of each layer and how it is shaped. Next, the number of parameters to be estimated in each layer will be computed.

Output Shape

Since we are using padding in the first layer (the argument padding =' same'), the output of the first layer will be exactly like its inputs which are $8 \times 8 \times 1$ images. Note that the number of filters in the first layer is 50; i.e. nf1 = 50, the filter size is 3 and stride is 1, so the output will be $8 \times 8 \times 50$. If we did not use padding, the output would be $6 \times 6 \times 50$ (Figure 1).

In the second convolutional layer, 150 filters are used with the same filter size and stride as those in Layer 1. As a result, the output of this layer is $6 \times 6 \times 150$.

A pooling layer with max operator is applied to the output of the second layer. Both the pooling size and stride are set to 2. This would shrink the size of the output layer to $3 \times 3 \times 150$.

After pooling, we flatten the layer, yielding a 1350×1 output. A dense layer with 100 units is added followed by an output layer with 10 units (10 classes).

Number of parameters

In the first convolutional layer, there are 50 filters. In each filter, there are 9 (3×3) parameters to estimate. Remember that filters are the same as units in a dense layer, so for each filter we have a bias parameter to be estimated. Consequently, there are 9×50 parameters in all 50 filters with 50 bias terms associated with these filters. This adds up to $9 \times 50 + 50 = 500$ parameters.

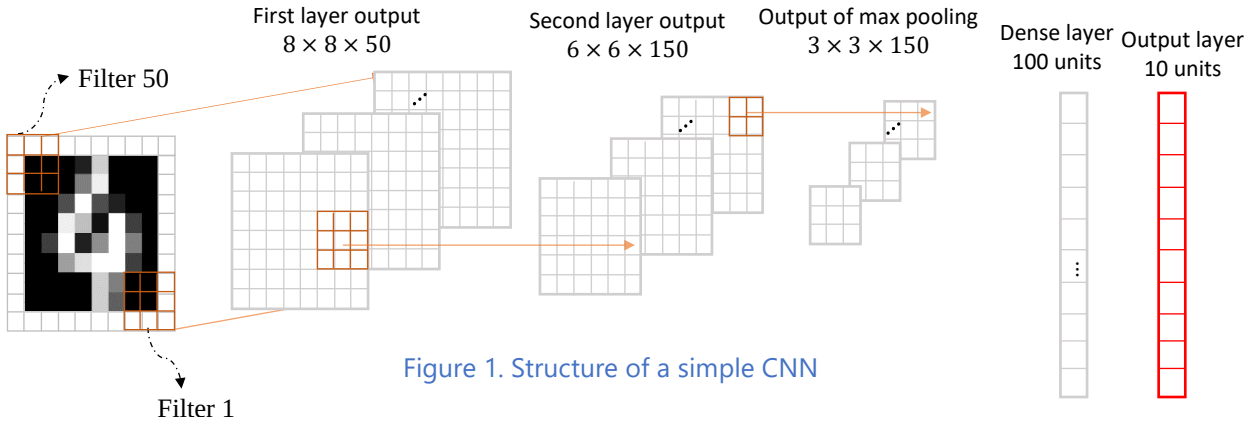


Figure 1. Structure of a simple CNN

The second convolutional layer has 150 filters. Computing the number of parameters for this layer is tricky. The output of the first layer is $8 \times 8 \times 50$. So, there are 50 activation maps. Each of these activation maps are fed to the second layer. In other words, each map is linked to all filters in the second layer just like the vanilla neural network where each output of the previous layer is connected to all units of the next layer. As a result, there are $50 \times 150 \times 3 \times 3$ parameters plus 150 bias terms giving 67650 parameters to be estimated for this layer.

Flatten and max pooling are just operators and do not add any parameters. However, max pooling shrinks the output of the second layer from $6 \times 6 \times 150$ to $3 \times 3 \times 150$. When flatten, this will be reordered to 1350×1.

The dense layer has 100 units. Each unit is connected to each of the 1350 units in the previously flattened layer. So, we will have $1350 \times 100 + 100 = 135100$ parameters in this layer.

In the last layer, all 10 units are connected to all 100 units in the previous layer. This gives $10 \times 100 + 10 = 1010$ parameters.

To get the total number of parameters just simply add the number of parameters in all layers. So, $1010 + 135100 + 67650 + 500 = 204260$ parameters in total!