Two Initial Methods for CNN

One is an initial method of data augmentation

second is an initial method of committe vote

By Li Yuan

The advantages of our new data augmenation

1. Use random multiple original inputs to create one new data (stochastic)

The advantages of our new data augmenation

1. Use random multiple original inputs to create one new data (stochastic)
1. Create new data based on each category instead of the whole new dataset to imporve feature extraction (classify)

The advantages of our new data augmenation

1. Use random multiple original inputs to create one new data (stochastic)
1. Create new data based on each category instead of the whole new dataset to imporve feature extraction (classify)
1. New data can be randomly selected to create another new data (self-adaptive)

The advantages of our new data augmenation

1. Use random multiple original inputs to create one new data (stochastic)
1. Create new data based on each category instead of the whole new dataset to imporve feature extraction (classify)
1. New data can be randomly selected to create another new data (self-adaptive)
1. Focus on feature extraction to create new data (committe vote)

The advantages of our new data augmenation

1. Use random multiple original inputs to create one new data (stochastic)
1. Create new data based on each category instead of the whole new dataset to imporve feature extraction (classify)
1. New data can be randomly selected to create another new data (self-adaptive)
1. Focus on feature extraction to create new data (committe vote)
1. Give multiple tune parameters to find best new data (tune parameter)

The disadvantaged of our new data augmentation

Slightly complex to implement

1. Seperate each class

The disadvantaged of our new data augmentation

Slightly complex to implement

1. Seperate each class

1. Create and tune paramaters on each class data

The steps used to test it on written digit number

1. Separate each class data and subset each class to reduce data size showing augmentation performance

The steps used to test it on written digit number

1. Separate each class data and subset each class to reduce data size showing augmentation performance
1. Generate new data based for each 10 classes then combine, shuffle them

The steps used to test it on written digit number

1. Separate each class data and subset each class to reduce data size showing augmentation performance
1. Generate new data based for each 10 classes then combine, shuffle them
1. train new data using CNN and evaluate results with some regularization methods

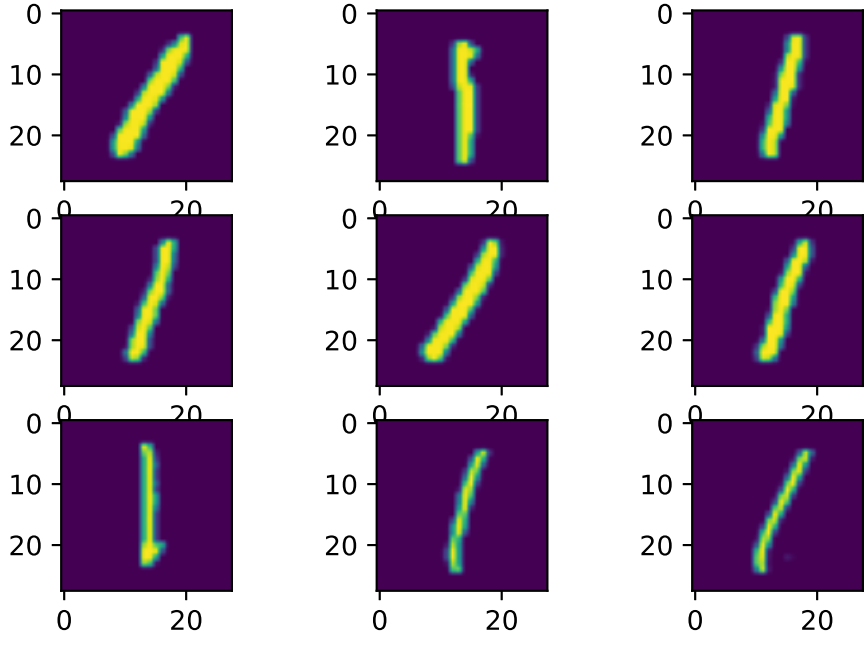
In [54]:

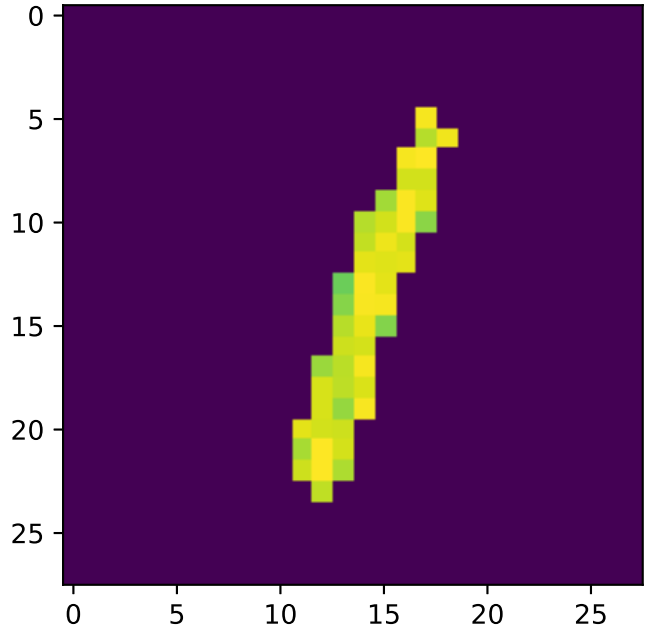
```
def generator(dataset, num = 0, S = 50, p = 0.25, thre = 140):
    '''
    by tuning paramaters we generate one new digit image
    '''
    cla = dataset
    np.random.shuffle(cla) # shuffle the class
    sub_index = np.random.choice(cla.shape[0], S, False) # random subset S data
    basis = cla[sub_index, :, :, :] # form the basis

    new = np.zeros_like(basis[1])
    for i in range(new.shape[0]):
        for j in range(new.shape[1]):
            count = []
            for q in range(S):
                if basis[q, i, j, 0] >= thre:
                    count.append(basis[q, i, j, 0])
            if len(count) >= S * p:
                new[i, j, 0] = np.mean(count)
    #plt.imshow(new)
    return new
```

In [155]:

```
generator1(num = 1, S = 9, p = 0.4, thre = 160)
```




0
5
10
15
20
25
0
5
10
15
20
25

Compare this new data augmentation with original data

Compare this new data augmentation with original data

In [61]:

```
print(train.shape, y.shape)
print(generated_train.shape, generated_y.shape)
```

```
(1195, 28, 28, 1) (1195,)
(8365, 28, 28, 1) (8365,)
```

Compare this new data augmentation with original data

In [61]:
```
print(train.shape, y.shape)
print(generated_train.shape, generated_y.shape)
```

```
(1195, 28, 28, 1) (1195,)
(8365, 28, 28, 1) (8365,)
```

In [146]:
```
model = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5), # dropout regularization
        layers.Dense(10, activation="softmax"),
    ]
)

model.summary()
```

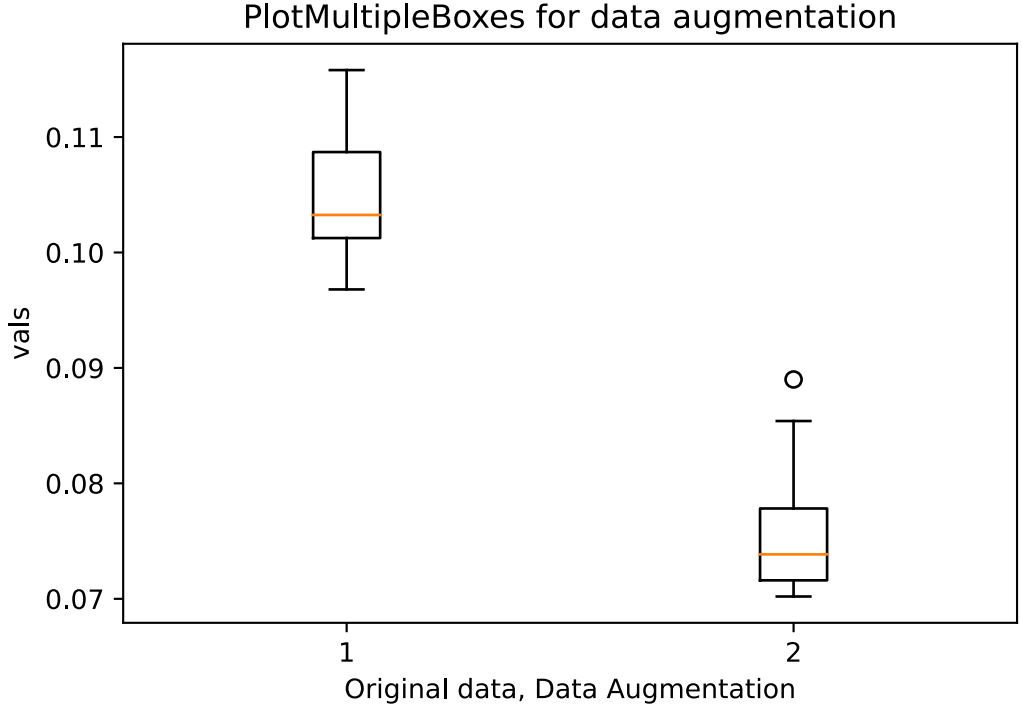
```
Model: "sequential_242"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_486 (Conv2D)          (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_486 (MaxPoolin (None, 13, 13, 32)        0
```

conv2d_487 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_487 (MaxPoolin	(None, 5, 5, 64)	0
flatten_243 (Flatten)	(None, 1600)	0
dropout_243 (Dropout)	(None, 1600)	0
dense_242 (Dense)	(None, 10)	16010

Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0

In [164]:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.boxplot(boxes)
ax.set_title('PlotMultipleBoxes for data augmentation')
ax.set_xlabel('Original data, Data Augmentation')
ax.set_ylabel('vals')
plt.show()
```



Initializa a new commmitte vote instead of mode or median vote

Use frequency weights on probability of committe vote

Initializa a new commmitte vote instead of mode or median vote

Use frequency weights on probability of committe vote

If the mode vote is greater than number of model * (tune parameter p), then we choose this vote as our prediction for each new test data

Initializa a new commmitte vote instead of mode or median vote

Use frequency weights on probability of committe vote

If the mode vote is greater than number of model * (tune parameter p), then we choose this vote as our prediction for each new test data

If the mode vote is less than number of model * (tune parameter p), then we use below method to choose vote

As an example, we randomly generate 10 models for predicting 10 digit numbers as probability 2D matrix

As an example, we randomly generate 10 models for predicting 10 digit numbers as probability 2D matrix

In [100]:
```
np.random.seed(123)
a = np.random.rand(10,10)
a
```

Out[100]:
```
array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
        0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752],
       [0.34317802, 0.72904971, 0.43857224, 0.0596779 , 0.39804426,
        0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759],
       [0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338,
        0.32295891, 0.36178866, 0.22826323, 0.29371405, 0.63097612],
       [0.09210494, 0.43370117, 0.43086276, 0.4936851 , 0.42583029,
        0.31226122, 0.42635131, 0.89338916, 0.94416002, 0.50183668],
       [0.62395295, 0.1156184 , 0.31728548, 0.41482621, 0.86630916,
        0.25045537, 0.48303426, 0.98555979, 0.51948512, 0.61289453],
       [0.12062867, 0.8263408 , 0.60306013, 0.54506801, 0.34276383,
        0.30412079, 0.41702221, 0.68130077, 0.87545684, 0.51042234],
       [0.66931378, 0.58593655, 0.6249035 , 0.67468905, 0.84234244,
        0.08319499, 0.76368284, 0.24366637, 0.19422296, 0.57245696],
       [0.09571252, 0.88532683, 0.62724897, 0.72341636, 0.01612921,
        0.59443188, 0.55678519, 0.15895964, 0.15307052, 0.69552953],
       [0.31876643, 0.6919703 , 0.55438325, 0.38895057, 0.92513249,
        0.84167   , 0.35739757, 0.04359146, 0.30476807, 0.39818568],
       [0.70495883, 0.99535848, 0.35591487, 0.76254781, 0.59317692,
        0.6917018 , 0.15112745, 0.39887629, 0.2408559 , 0.34345601]])
```

As an example, we randomly generate 10 models for predicting 10 digit numbers as probability 2D matrix

In [100]:
```
np.random.seed(123)
a = np.random.rand(10,10)
a
```

Out[100]:
```
array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
        0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752],
       [0.34317802, 0.72904971, 0.43857224, 0.0596779 , 0.39804426,
        0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759],
       [0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338,
        0.32295891, 0.36178866, 0.22826323, 0.29371405, 0.63097612],
       [0.09210494, 0.43370117, 0.43086276, 0.4936851 , 0.42583029,
        0.31226122, 0.42635131, 0.89338916, 0.94416002, 0.50183668],
       [0.62395295, 0.1156184 , 0.31728548, 0.41482621, 0.86630916,
        0.25045537, 0.48303426, 0.98555979, 0.51948512, 0.61289453],
       [0.12062867, 0.8263408 , 0.60306013, 0.54506801, 0.34276383,
        0.30412079, 0.41702221, 0.68130077, 0.87545684, 0.51042234],
       [0.66931378, 0.58593655, 0.6249035 , 0.67468905, 0.84234244,
        0.08319499, 0.76368284, 0.24366637, 0.19422296, 0.57245696],
       [0.09571252, 0.88532683, 0.62724897, 0.72341636, 0.01612921,
        0.59443188, 0.55678519, 0.15895964, 0.15307052, 0.69552953],
       [0.31876643, 0.6919703 , 0.55438325, 0.38895057, 0.92513249,
        0.84167   , 0.35739757, 0.04359146, 0.30476807, 0.39818568],
       [0.70495883, 0.99535848, 0.35591487, 0.76254781, 0.59317692,
        0.6917018 , 0.15112745, 0.39887629, 0.2408559 , 0.34345601]])
```

Vote Results and probability for each voted digit number then compute mean of each list

As an example, we randomly generate 10 models for predicting 10 digit numbers as probability 2D matrix

In [100]:
```
np.random.seed(123)
a = np.random.rand(10,10)
a
```

Out[100]:
```
array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
        0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752],
       [0.34317802, 0.72904971, 0.43857224, 0.0596779 , 0.39804426,
        0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759],
       [0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338,
        0.32295891, 0.36178866, 0.22826323, 0.29371405, 0.63097612],
       [0.09210494, 0.43370117, 0.43086276, 0.4936851 , 0.42583029,
        0.31226122, 0.42635131, 0.89338916, 0.94416002, 0.50183668],
       [0.62395295, 0.1156184 , 0.31728548, 0.41482621, 0.86630916,
        0.25045537, 0.48303426, 0.98555979, 0.51948512, 0.61289453],
       [0.12062867, 0.8263408 , 0.60306013, 0.54506801, 0.34276383,
        0.30412079, 0.41702221, 0.68130077, 0.87545684, 0.51042234],
       [0.66931378, 0.58593655, 0.6249035 , 0.67468905, 0.84234244,
        0.08319499, 0.76368284, 0.24366637, 0.19422296, 0.57245696],
       [0.09571252, 0.88532683, 0.62724897, 0.72341636, 0.01612921,
        0.59443188, 0.55678519, 0.15895964, 0.15307052, 0.69552953],
       [0.31876643, 0.6919703 , 0.55438325, 0.38895057, 0.92513249,
        0.84167   , 0.35739757, 0.04359146, 0.30476807, 0.39818568],
       [0.70495883, 0.99535848, 0.35591487, 0.76254781, 0.59317692,
        0.6917018 , 0.15112745, 0.39887629, 0.2408559 , 0.34345601]])
```

Vote Results and probability for each voted digit number then compute mean of each list

In [135]:
```
print("Vote results:", vote, "\n")
pri_avg(ind_new)
```

```
Vote results: [6. 5. 1. 8. 7. 8. 4. 1. 4. 1.]

1 :  [0.8494317940777896, 0.8853268262751396, 0.9953584820340174] / 3 = 0.
9100390341289822

4 :  [0.8423424376202573, 0.9251324896139861] / 2 = 0.8837374636171217

5 :  [0.7379954057320357] / 1 = 0.7379954057320357

6 :  [0.9807641983846155] / 1 = 0.9807641983846155

7 :  [0.985559785610705] / 1 = 0.9855597856107050

8 :  [0.9441600182038796, 0.8754568417951749] / 2 = 0.9098084299995273
```

As an example, we randomly generate 10 models for predicting 10 digit numbers as probability 2D matrix

In [100]:
```
np.random.seed(123)
a = np.random.rand(10,10)
a
```

Out[100]:
```
array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
        0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752],
       [0.34317802, 0.72904971, 0.43857224, 0.0596779 , 0.39804426,
        0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759],
       [0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338,
        0.32295891, 0.36178866, 0.22826323, 0.29371405, 0.63097612],
       [0.09210494, 0.43370117, 0.43086276, 0.4936851 , 0.42583029,
        0.31226122, 0.42635131, 0.89338916, 0.94416002, 0.50183668],
       [0.62395295, 0.1156184 , 0.31728548, 0.41482621, 0.86630916,
        0.25045537, 0.48303426, 0.98555979, 0.51948512, 0.61289453],
       [0.12062867, 0.8263408 , 0.60306013, 0.54506801, 0.34276383,
        0.30412079, 0.41702221, 0.68130077, 0.87545684, 0.51042234],
       [0.66931378, 0.58593655, 0.6249035 , 0.67468905, 0.84234244,
        0.08319499, 0.76368284, 0.24366637, 0.19422296, 0.57245696],
       [0.09571252, 0.88532683, 0.62724897, 0.72341636, 0.01612921,
        0.59443188, 0.55678519, 0.15895964, 0.15307052, 0.69552953],
       [0.31876643, 0.6919703 , 0.55438325, 0.38895057, 0.92513249,
        0.84167   , 0.35739757, 0.04359146, 0.30476807, 0.39818568],
       [0.70495883, 0.99535848, 0.35591487, 0.76254781, 0.59317692,
        0.6917018 , 0.15112745, 0.39887629, 0.2408559 , 0.34345601]])
```

Vote Results and probability for each voted digit number then compute mean of each list

In [135]:
```
print("Vote results:", vote, "\n")
pri_avg(ind_new)
```

```
Vote results: [6. 5. 1. 8. 7. 8. 4. 1. 4. 1.]

1 :  [0.8494317940777896, 0.8853268262751396, 0.9953584820340174] / 3 = 0.
9100390341289822

4 :  [0.8423424376202573, 0.9251324896139861] / 2 = 0.8837374636171217

5 :  [0.7379954057320357] / 1 = 0.7379954057320357

6 :  [0.9807641983846155] / 1 = 0.9807641983846155

7 :  [0.985559785610705] / 1 = 0.9855597856107050

8 :  [0.9441600182038796, 0.8754568417951749] / 2 = 0.9098084299995273
```

Compute average probability of each digit multiplied by their weights

As an example, we randomly generate 10 models for predicting 10 digit numbers as probability 2D matrix

In [100]:
```
np.random.seed(123)
a = np.random.rand(10,10)
a
```

Out[100]:
```
array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
        0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752],
       [0.34317802, 0.72904971, 0.43857224, 0.0596779 , 0.39804426,
        0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759],
       [0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338,
        0.32295891, 0.36178866, 0.22826323, 0.29371405, 0.63097612],
       [0.09210494, 0.43370117, 0.43086276, 0.4936851 , 0.42583029,
        0.31226122, 0.42635131, 0.89338916, 0.94416002, 0.50183668],
       [0.62395295, 0.1156184 , 0.31728548, 0.41482621, 0.86630916,
        0.25045537, 0.48303426, 0.98555979, 0.51948512, 0.61289453],
       [0.12062867, 0.8263408 , 0.60306013, 0.54506801, 0.34276383,
        0.30412079, 0.41702221, 0.68130077, 0.87545684, 0.51042234],
       [0.66931378, 0.58593655, 0.6249035 , 0.67468905, 0.84234244,
        0.08319499, 0.76368284, 0.24366637, 0.19422296, 0.57245696],
       [0.09571252, 0.88532683, 0.62724897, 0.72341636, 0.01612921,
        0.59443188, 0.55678519, 0.15895964, 0.15307052, 0.69552953],
       [0.31876643, 0.6919703 , 0.55438325, 0.38895057, 0.92513249,
        0.84167   , 0.35739757, 0.04359146, 0.30476807, 0.39818568],
       [0.70495883, 0.99535848, 0.35591487, 0.76254781, 0.59317692,
        0.6917018 , 0.15112745, 0.39887629, 0.2408559 , 0.34345601]])
```

Vote Results and probability for each voted digit number then compute mean of each list

In [135]:
```
print("Vote results:", vote, "\n")
pri_avg(ind_new)
```

```
Vote results: [6. 5. 1. 8. 7. 8. 4. 1. 4. 1.]

1 :  [0.8494317940777896, 0.8853268262751396, 0.9953584820340174] / 3 = 0.
9100390341289822

4 :  [0.8423424376202573, 0.9251324896139861] / 2 = 0.8837374636171217

5 :  [0.7379954057320357] / 1 = 0.7379954057320357

6 :  [0.9807641983846155] / 1 = 0.9807641983846155

7 :  [0.985559785610705] / 1 = 0.9855597856107050

8 :  [0.9441600182038796, 0.8754568417951749] / 2 = 0.9098084299995273
```

Compute average probability of each digit multiplied by their weights

In [121]:
```
pri_mul(ind_mean, ind_num)
```

```
1 :  0.9100390341289822 * (3 / 10) = 0.2730117102386947
```

4 : 0.8837374636171217 * (2 / 10) = 0.1767474927234243

5 : 0.7379954057320357 * (1 / 10) = 0.0737995405732036

6 : 0.9807641983846155 * (1 / 10) = 0.0980764198384615

7 : 0.9855597856107050 * (1 / 10) = 0.0985559785610705

8 : 0.9098084299995273 * (2 / 10) = 0.1819616859999054

In [139]:

```
def committe_vote(p, **kwargs):
    '''
    redefine a new committe vote modality
    '''
    con = []
    for key, value in kwargs.items():
        value = np.expand_dims(value, 0)
        con.append(value)
    new = np.concatenate(con, axis = 0)
    num_test = new.shape[1]
    num_model = new.shape[0]
    num_class = new.shape[2]
    pred = np.empty(shape=num_test)

    for i in range(num_test):
        ma = new[:, i, :]
        vote = np.empty(num_model)
        ind_pro = {0: [], 1: [],2: [],3: [],4: [],5: [],6: [],7: [],8: [],9: []}
        for j in np.arange(num_model):
            vote[j] = np.argmax(ma[j, :])
            ind_pro.get(vote[j]).append(np.amax(ma[j, :]))
        m, n = stats.mode(vote, axis=None)
        if n >= num_model * p:
            pred[i] = m
        else:
            ind_new = {}
            for key, value in ind_pro.items():
                if value != []:
                    ind_new[key] = value
            ind_mean = {}
            ind_num = {}
            for key, value in ind_new.items():
                ind_mean[key] = np.mean(value)
                ind_num[key] = len(value)
            ind_final = {}
            for key in ind_new:
                ind_final[key] = ind_mean[key] * ind_num[key] / num_model
            need = max(ind_final.values())
```

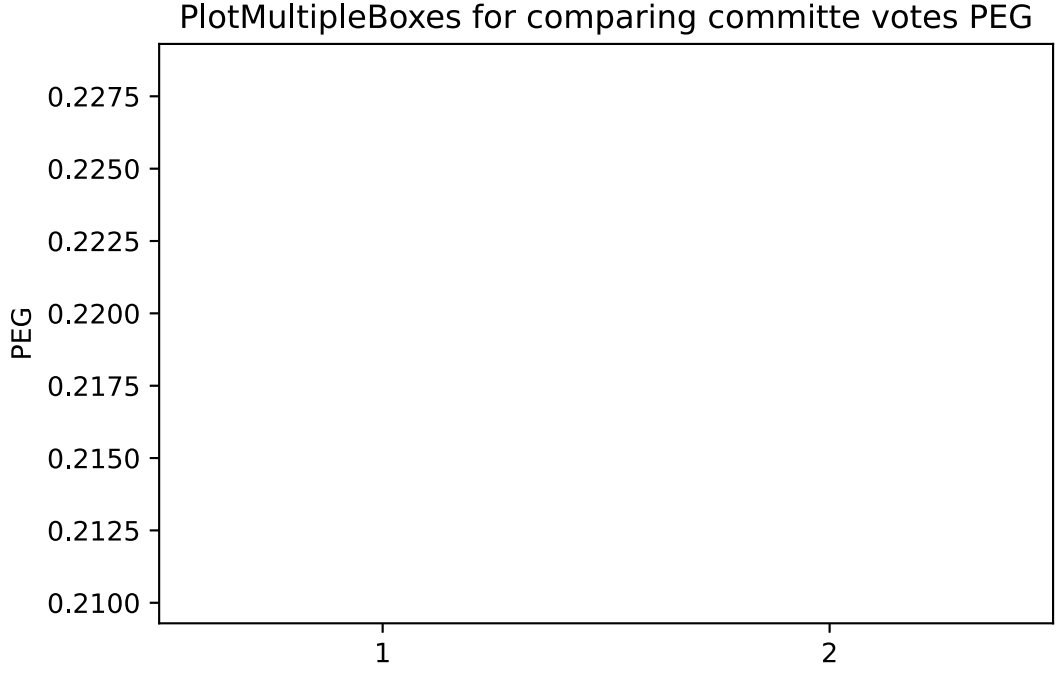
```
            for key, value in ind_final.items():
                if value == need:
                    pred[i] = key
    return pred
```

After using 10 restarts and 10 models as a committe, we used box plot to compare their performance.

After using 10 restarts and 10 models as a committe, we used box plot to compare their performance.

In [163]:

```
boxes2 = [PEG_median, PEG_pro_wei]
fig = plt.figure()
ax = fig.add_subplot(111)
ax.boxplot(boxes2)
ax.set_title('PlotMultipleBoxes for comparing committe votes PEG')
ax.set_ylabel('PEG')
plt.show()
```



What have we learned?

1. By doing this research project, I have learned how to design our initial algorithm to imporve current algorithms and think to push the boundary of current research work.

2. By conducting a self-paced research, it encourages me to think independently and learn new things

3. Use boxes plot or other methods to test our initial method

What have we learned?

1. By doing this research project, I have learned how to design our initial algorithm to imporve current algorithms and think to push the boundary of current research work.

2. By conducting a self-paced research, it encourages me to think independently and learn new things

3. Use boxes plot or other methods to test our initial method

Thanks