2.

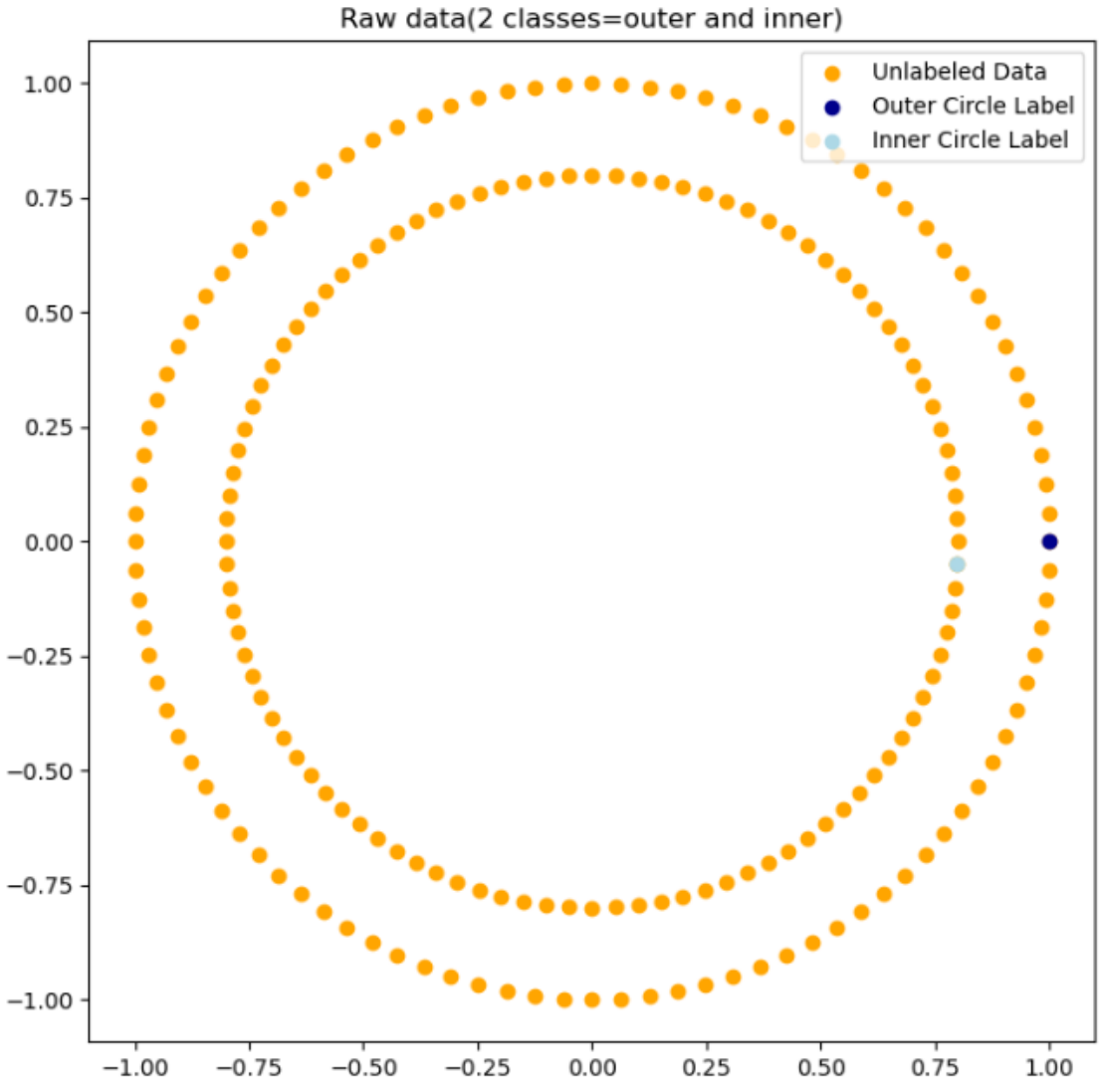
Question 2

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

n_samples = 200 # total number of points in both inner and outer circles combined
X, y = make_circles(n_samples=n_samples, shuffle=False)
# X is data points coordinates[feature matrix], y is the labels of data points.
# Initially all the data points are labelled with '0's and '1's [100 each].
# we create a numpy array with -1 for all 200 data points, -1 indicate the data point
# is unlabelled.
labels = np.full(n_samples, -1.0, dtype=float)
# we will assign labels to data points based on the question:
#0 --> outer circles first datapoint label.
#1 --> inner circles last data point label
outer, inner = 0, 1
# Label some points as the "outer" and "inner" circles for label propagation
labels[0] = outer
labels[-1] = inner
```

```
print(f"""Analysis on indexing of labels:
The first element is: labels[0] --> {labels[0]}
or labels[200] --> {labels[-200]}
and the last element is: labels[-1] --> {labels[-1]} or
labels[199] --> {labels[199]}""")
```

```
Analysis on indexing of labels:
The first element is: labels[0] --> 0.0
or labels[200] --> 0.0
and the last element is: labels[-1] --> 1.0 or
labels[199] --> 1.0
```

```python
# Plotting the generated circles
plt.figure(figsize=(8, 8))
# X[:,0] --> x coordinate, X[:,1] --> y coordinate
plt.scatter(X[:, 0], X[:, 1], color='orange', label='Unlabeled Data')
# plot the outer labelled data point by calling the 0th row in X matrix with dark blue
plt.scatter(X[0, 0], X[0, 1], color='darkblue', label='Outer Circle Label')
#plot the inner labelled data point by calling the -1st row or 199th row in X matrix with light blue
plt.scatter(X[199, 0], X[199, 1], color='lightblue', label='Inner Circle Label')
# add title, legend to the plot
plt.title("Raw data(2 classes=outer and inner)")
plt.legend(loc = 'upper right')
plt.axis('equal')  # Equal scaling for both axes
plt.show()
```



Implement label spreading: ¶

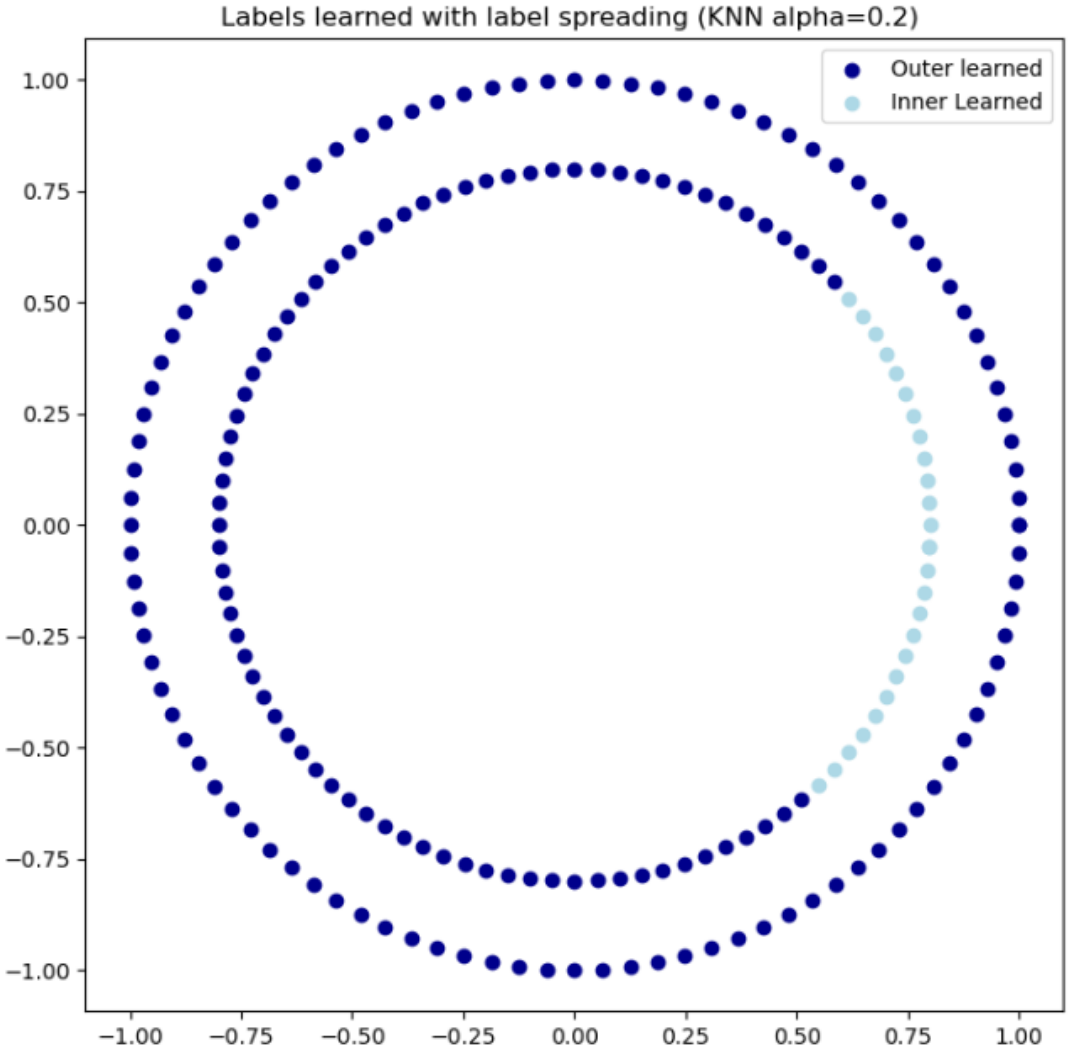
While implementing label spreading, I want to experiment, observe and analyse the affects of decay parameter in semi-supervised labelling.

```
]: from sklearn.semi_supervised import LabelSpreading
# Apply Label Spreading
#alpha = 0.2
label_spread = LabelSpreading(kernel='knn', alpha=0.2) #using knn
label_spread.fit(X, labels) # model fitting

# Predict the labels for all points
predicted_labels = label_spread.transduction_
print(f"The original labels are: \n {labels}")
print(f"The predicted labels are: \n {predicted_labels}")
```

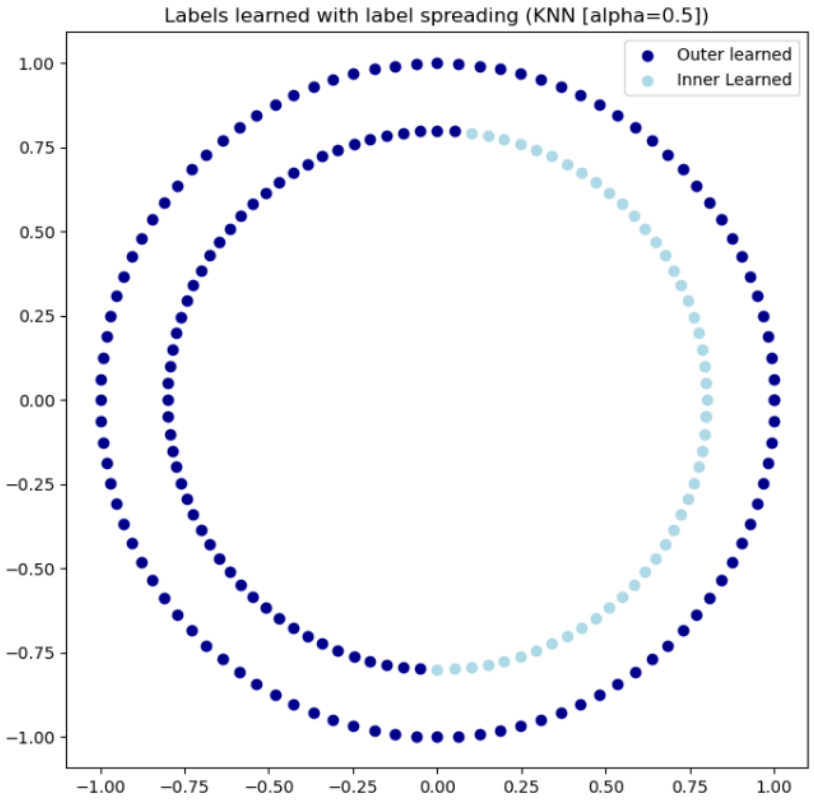
```
The original labels are:
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
 -1.  1.]
The predicted labels are:
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1.]
```

For plotting the label spreading results, i came up with this approach to color the data points based on its predicted label values. for i in num_samples: if predicted labels[i] == 0, color with dark blue, elif predicted labels[i] == 1 color with light blue, else color with orange

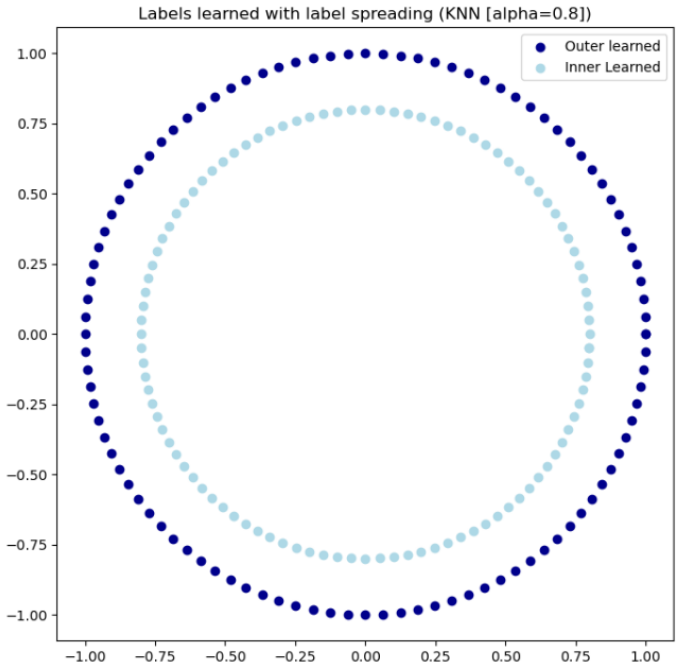
```python
# Plotting based on predicted labels
plt.figure(figsize=(8, 8))
for i in range(n_samples):
    if predicted_labels[i] == 0:
        plt.scatter(X[i, 0], X[i, 1], color='darkblue')
    elif predicted_labels[i] == 1:
        plt.scatter(X[i, 0], X[i, 1], color='lightblue')
    else:
        plt.scatter(X[i, 0], X[i, 1], color='orange', label = 'Unlabelled')

# I choose first and last data point to assign title to the plotting.
plt.scatter(X[0, 0], X[0, 1], color='darkblue', label='Outer learned')
plt.scatter(X[-1, 0], X[-1, 1], color='lightblue', label = 'Inner Learned')

# Add plot title, labels, legend, and show
plt.title("Labels learned with label spreading (KNN alpha=0.2) ")
plt.legend(loc='upper right')
plt.axis('equal')
plt.show()
```



```python
#alpha = 0.5
label_spread = LabelSpreading(kernel='knn', alpha=0.5) #using knn
label_spread.fit(X, labels) # model fitting
# Predict the labels for all points
predicted_labels = label_spread.transduction_
# Plotting based on predicted labels
plt.figure(figsize=(8, 8))
for i in range(n_samples):
    if predicted_labels[i] == 0:
        plt.scatter(X[i, 0], X[i, 1], color='darkblue')
    elif predicted_labels[i] == 1:
        plt.scatter(X[i, 0], X[i, 1], color='lightblue')
    else:
        plt.scatter(X[i, 0], X[i, 1], color='orange', label = 'Unlabelled')
# I choose first and last data point to assign title to the plotting.
plt.scatter(X[0, 0], X[0, 1], color='darkblue', label='Outer learned')
plt.scatter(X[-1, 0], X[-1, 1], color='lightblue', label = 'Inner Learned')

# Add plot title, labels, legend, and show
plt.title("Labels learned with label spreading (KNN [alpha=0.5])")
plt.legend(loc='upper right')
plt.axis('equal')
plt.show()
```



```python
#alpha = 0.8
label_spread = LabelSpreading(kernel='knn', alpha=0.8) #using knn
label_spread.fit(X, labels) # model fitting

# Predict the labels for all points
predicted_labels = label_spread.transduction_

# Plotting based on predicted labels
plt.figure(figsize=(8, 8))
for i in range(n_samples):
    if predicted_labels[i] == 0:
        plt.scatter(X[i, 0], X[i, 1], color='darkblue')
    elif predicted_labels[i] == 1:
        plt.scatter(X[i, 0], X[i, 1], color='lightblue')
    else:
        plt.scatter(X[i, 0], X[i, 1], color='orange', label = 'Unlabelled')

# I choose first and last data point to assign title to the plotting.
plt.scatter(X[0, 0], X[0, 1], color='darkblue', label='Outer learned')
plt.scatter(X[-1, 0], X[-1, 1], color='lightblue', label = 'Inner Learned')

# Add plot title, labels, legend, and show
plt.title("Labels learned with label spreading (KNN [alpha=0.8])")
plt.legend(loc='upper right')
plt.axis('equal')
plt.show()
```



We can observe the value of decay parameter effects the semi-supervised labelling significantly.

As we increase the value of alpha, the labels are predicted more accurately.

At alpha =0.8 we can see that Label Spreading can propagate labels correctly around the circle. This supports the definition of decay parameter which is alpha controls how much of label info spreads from neighbors in each iteration.

3.

Question 3

```
# I am using the same sample code given in the question to load the dataset.
import numpy as np
from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import accuracy_score, confusion_matrix # performance metrics as mentioned in the question
# Load the digit dataset
digits = datasets.load_digits()
rng = np.random.RandomState(0)
indices = np.arange(len(digits.data))
rng.shuffle(indices)
X = digits.data[indices[:330]] # flattened array of 2d image with its values of pixels in an image
y = digits.target[indices[:330]] # number labels of the 1d arrays in X matrix
images = digits.images[indices[:330]] # This represent the image in 2d before flattening.
n_total_samples = len(y)
print(f"The number of samples in the dataset after indicing is {n_total_samples}")
```

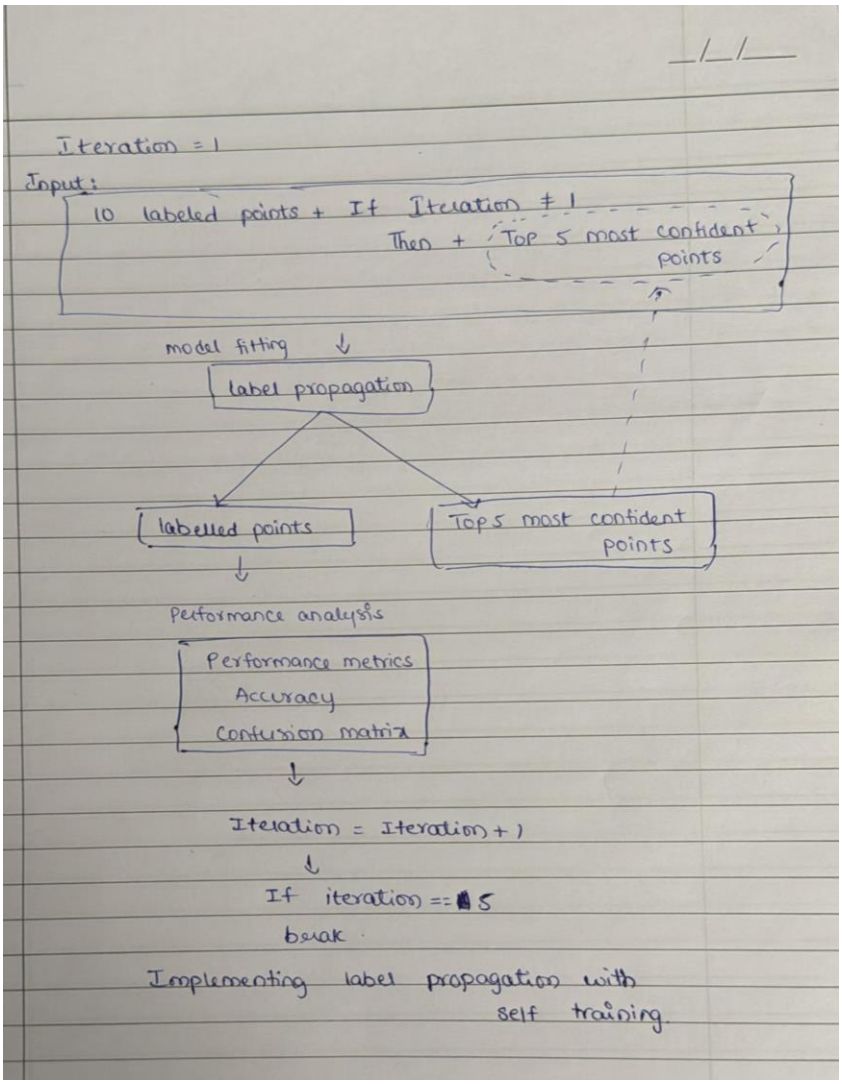
```
The number of samples in the dataset after indicing is 330
```

```
# I will create a new labels array with -1 as value for all 330 samples.
labels = np.full(n_total_samples, -1.0, dtype=float)
n_labeled_points = 10

init_indices = np.arange(n_labeled_points)
labels[init_indices] = y[init_indices]  #  learning a label propagation model with only
                                                                #10 labeled points,


print(f"The first 20 labels in the labels array is {labels[:20]}")
```

```
The first 20 labels in the labels array is [ 2.  8.  2.  6.  6.  7.  1.  9.  8.  5. -1. -1. -1. -1. -1. -1. -1. -1.
 -1. -1.]
```

Since we have shuffled the data, when we are considering the first 10 labels to use for fitting the label spreading model, we are providing label information to partial classes in the whole dataset, for suppose in this case we have label information to classes: 1, 2, 5, 6, 7, 8, 9. So in this process we are not providing any label information for classes 0, 3, 4.

lets see how the model labels for these classes using confusion metrics in later stage.



I am going to follow this architecture to implement label propagation with self training.

```python
# I am writing this cell to check if i am able to extract the indices of top 5 most confident pointns
model = LabelSpreading(kernel='knn', alpha=0.8)
model.fit(X, labels)

# Predict scores for all points to find the most confident predictions
pred_scores = model.predict_proba(X)
print(f'The first 5 samples prediction scores from label spreading {pred_scores[:5]}')
print('-'*40)
unlabeled_indices = np.where(labels == -1)[0]
# I am putting the axis=1, so i can find the maximum prediction score of the model for each sample
confidence_scores = pred_scores[unlabeled_indices].max(axis=1)
# To choose the top 5 confidence scores by label spreading, I am sorting it ascending order and choose the final 5 indices
top_5_confident_indices = np.argsort(confidence_scores)[-5:]
top_5 = unlabeled_indices[top_5_confident_indices]
print(f'the top 5 unlabeled indices havign the best confidence scores are: {top_5}')
```

```
The first 5 samples prediction scores from label spreading [[1.64211059e-04 9.99452055e-01 2.75562541e-08 9.70591130e-05
  2.00481223e-06 2.73054288e-04 1.15883010e-05]
 [1.69610421e-02 2.37612363e-03 5.47074956e-05 8.75276708e-04
  2.23769888e-03 9.77244465e-01 2.50686683e-04]
 [1.18952761e-03 9.96913450e-01 1.36665270e-07 1.35200559e-04
  1.09579305e-05 1.60472031e-03 1.46006622e-04]
 [1.50756869e-06 8.15767391e-07 1.83111925e-09 9.99988669e-01
  2.13887417e-07 8.44290260e-06 3.48993228e-07]
 [2.43171107e-06 1.35235316e-06 3.25772287e-09 9.99981634e-01
  3.51870571e-07 1.36478391e-05 5.79351127e-07]]
----------------------------------------
the top 5 unlabeled indices havign the best confidence scores are: [ 45 307 300 126  83]
```

We can see that I am able to select the top 5 most confident points to label, so i am gonna use this code in a while loop and iterate it till I train the model with 30 labeled examples

```python
labels = np.full(n_total_samples, -1.0, dtype=float)
n_labeled_points = 10

init_indices = np.arange(n_labeled_points)
labels[init_indices] = y[init_indices]  #  learning a label propagation model with only
                                                    #10 labeled points
while n_labeled_points <=30:

    # I am choosing alpha = 0.8 which i observed as the optimal value of alpha from 2nd question
    model = LabelSpreading(kernel='knn', alpha=0.8)
    model.fit(X, labels)
    print(f'no. of label points considered for model fitting are: {n_labeled_points}')

    # Report results after each iteration
    pred_labels = model.predict(X)
    # Performance metrics
    acc = accuracy_score(y, pred_labels)
    #print(f'no. of label points considered for model fitting are: {n_labeled_points}')
    print(f"Accuracy when {n_labeled_points} are considered is: {acc}")
    print("Confusion Matrix:")
    print(conf_matrix)
    print("-"*70)
    #finding the top 5 most confident points and adding them to label dataset.
    pred_scores = model.predict_proba(X)
    unlabeled_indices = np.where(labels == -1)[0]
    # I am putting the axis=1, so i can find the maximum prediction score of the model for each sample
    confidence_scores = pred_scores[unlabeled_indices].max(axis=1)
    # To choose the top 5 confidence scores by label spreading, I am sorting it ascending order and choose the final 5 indices
    top_5_confident_indices = np.argsort(confidence_scores)[-5:]
    top_5 = unlabeled_indices[top_5_confident_indices]
    # Add newly labeled points to the labeled set
    labels[top_5] = y[top_5]
    print(f'the class labels added based on top 5 indices are: {labels[top_5]}')
    n_labeled_points += 5
```

```
no. of label points considered for model fitting are: 10
Accuracy when 10 are considered is: 0.6121212121212121
Confusion Matrix:
[[ 0 24  0  0  0  0  0  0  0  0]
 [ 0  1  5  0  0  0  9  3 12  0]
 [ 0  0 31  0  0  0  0  2  0  0]
 [ 0  0 26  0  0  0  0  0  1  1]
 [ 0  0  0  0  0  0  0 27  0  0]
 [ 0  0  0  0  0 25  0  0  0 11]
 [ 0  0  0  0  0  0 42  0  0  0]
 [ 0  0  0  0  0  0  0 37  0  0]
 [ 0  1  0  0  0  0  0  0 34  0]
 [ 0  0  0  0  0  3  0  2  1 32]]
--------------------------------------------------------------------------
the class labels added based on top 5 indices are: [0. 0. 0. 0. 0.]
no. of label points considered for model fitting are: 15
Accuracy when 15 are considered is: 0.6757575757575758
Confusion Matrix:
[[24  0  0  0  0  0  0  0  0  0]
 [ 0  1  5  0  0  0  9  3 12  0]
 [ 1  0 30  0  0  0  0  2  0  0]
 [ 0  0 26  0  0  0  0  0  1  1]
 [ 0  0  0  0  0  0  0 27  0  0]
 [ 0  0  0  0  0 25  0  0  0 11]
 [ 0  0  0  0  0  0 42  0  0  0]
 [ 0  0  0  0  0  0  0 37  0  0]
 [ 0  1  0  0  0  0  0  0 34  0]
 [ 1  0  0  0  0  3  0  3  1 30]]
--------------------------------------------------------------------------
```

```
the class labels added based on top 5 indices are: [0. 0. 0. 0. 0.]
no. of label points considered for model fitting are: 20
Accuracy when 20 are considered is: 0.6757575757575758
Confusion Matrix:
[[24  0  0  0  0  0  0  0  0  0]
 [ 0  1  5  0  0  0  9  3 12  0]
 [ 1  0 30  0  0  0  0  2  0  0]
 [ 0  0 26  0  0  0  0  0  1  1]
 [ 0  0  0  0  0  0  0 27  0  0]
 [ 0  0  0  0  0 25  0  0  0 11]
 [ 0  0  0  0  0  0 42  0  0  0]
 [ 0  0  0  0  0  0  0 37  0  0]
 [ 0  1  0  0  0  0  0  0 34  0]
 [ 1  0  0  0  0  3  0  3  1 30]]
----------------------------------------------------------------------
the class labels added based on top 5 indices are: [0. 0. 0. 0. 0.]
no. of label points considered for model fitting are: 25
Accuracy when 25 are considered is: 0.5787878787878787
Confusion Matrix:
[[24  0  0  0  0  0  0  0  0  0]
 [ 0  1  5  0  0  0  9  3 12  0]
 [ 3  0 28  0  0  0  0  2  0  0]
 [ 1  0 26  0  0  0  0  0  1  0]
 [ 0  0  0  0  0  0  0 27  0  0]
 [11  0  0  0  0 25  0  0  0  0]
 [ 0  0  0  0  0  0 42  0  0  0]
 [ 0  0  0  0  0  0  0 37  0  0]
 [ 0  1  0  0  0  0  0  0 34  0]
 [32  0  0  0  0  3  0  2  1  0]]
----------------------------------------------------------------------

the class labels added based on top 5 indices are: [0. 0. 0. 0. 0.]
no. of label points considered for model fitting are: 30
Accuracy when 30 are considered is: 0.5787878787878787
Confusion Matrix:
[[24  0  0  0  0  0  0  0  0  0]
 [ 0  1  5  0  0  0  9  3 12  0]
 [ 3  0 28  0  0  0  0  2  0  0]
 [ 1  0 26  0  0  0  0  0  1  0]
 [ 0  0  0  0  0  0  0 27  0  0]
 [11  0  0  0  0 25  0  0  0  0]
 [ 0  0  0  0  0  0 42  0  0  0]
 [ 0  0  0  0  0  0  0 37  0  0]
 [ 0  1  0  0  0  0  0  0 34  0]
 [32  0  0  0  0  3  0  2  1  0]]
----------------------------------------------------------------------
the class labels added based on top 5 indices are: [6. 0. 0. 0. 0.]
```

As we discussed earlier, from the confusion matrix obtained on label spreading with 10 labels, theres clearly no label predicted for number classes 0, 3, 4. This can be understood by zero predictions made for these classes in the confusion matrix. But interestingly when we are finding the top 5 most confident points by the model, all the top 5 confident points were having the original labels of 0.

```
#Details of the labled samples at the end of training with 30 samples.
for i in range(10):
    print(f' for class {i} indices considered are {np.where(labels == i)[0]}')
```

```
for class 0 indices considered are [ 17  38  45  54  55  60  83 105 126 165 172 184 188 194 201 208 237 259
269 300 304 307 310 328]
for class 1 indices considered are [6]
for class 2 indices considered are [0 2]
for class 3 indices considered are []
for class 4 indices considered are []
for class 5 indices considered are [9]
for class 6 indices considered are [  3   4 139]
for class 7 indices considered are [5]
for class 8 indices considered are [1 8]
for class 9 indices considered are [7]
```

The above is the dataset distribution when considering 30 samples. We can see at the end class 3 and class 4 do not have any samples, so there is no predictions made for these 2 classes. This is clearly case of imbalanced dataset. Since the majority of the labeled samples are 0s, the predicted labels were mostly 0. At 30 labeled sampples the number of samples predicted as 0 is 24 + 3 + 1+ 11+ 32 = 71 while for number class 6, the samples predicted as 6 is 9+42 = 51. This explains that if we give a biased dataset to train a model, the results will also be biased to them.

The accuracy has increased from 61.25% to 67.57% and dropped down to 57.87%. This could mainly be due to the imbalanced dataset we are feeding to the model. In addition, 67.57% may sound average performance, but model did not predict for class 3, 4. So, we need to consider other performance metrics like weighted accuracy, class wise precision, recall for better performance analysis.

Question 4:

After exploring the given repository [https://github.com/tkipf/pygcn], I understood the working of GCN and the details of Cora Dataset.

Before diving into the model training, lets explore the dataset and its contents which I Understood from the GitHub repository contents.

The **Cora dataset** consists of Machine Learning papers. These papers are classified into one of the following **seven classes**: Case_Based, Genetic_Algorithms, Neural_Networks, Probabilistic_Methods, Reinforcement_Learning, Rule_Learning, Theory.

These research papers are selected in way that in final corpus every paper cites or is cited by atleast one another paper. There are a total of 2708 papers in the whole corpus.

After stemming and removing stopwords we were left with a vocabulary of size 1433 unique words. All words with document frequency less than 10 were removed.

So basically every research article can be expressed as 1433 length one dimensional vector with its values either 1 or 0[where 1 if the unique word exists in the article and 0 otherwise] .

Coming to the source code provided in the repository, I have observed in utils.py that the graph is constructed is constructed in such a way that the nodes represent research articles while the edges represent the citations. The edges are directed where paper having the citation is directed to paper being cited.

Hyper parameters used:

I am using most of the default hyper parameters defined in the source code, though I made minor changes for size of training dataset in reference to given question.

S. No.	Hyper parameter	Parameter used
1.	Seed value for random generator of kernels	42
2.	Epochs for training	200
3.	Learning rate [lr]	0.01
4.	Weight decay	5e-4
5.	Number of hidden layers	16
6.	Dropout percentage[Regularization method]	0.5
7.	Train val test split	300: 300: 2100

Note: Initially the train val test split is 200:300: 1000. That is train is range(200), validation is range (200:500) and testing is range (500, 1500). Since the source code is not using the complete dataset I have modified the train validation and test dataset size for complete use of dataset. Additionally, as mentioned in question, I am going to use [60, 120, 180, 240, 300] nodes from train set by indexing, this way I can set the number of labelled nodes as per my requirement.

Below is the source code which I modified from utils.py

```
idx_train = range(200)
idx_val = range(200, 500)
idx_test = range(500, 1500)
```

To

```
idx_train = range(300)
idx_val = range(300, 600)
idx_test = range(600, 2700)
```

Changes made in train.py:

```python
# Load data
adj, features, labels, idx_train_full, idx_val, idx_test = load_data()

# labeled nodes considered for GCN training based on the question
labeled_nodes = [60, 120, 180, 240, 300]

# Looping through each labeled node count and evaluating the model
for n_labeled in labeled_nodes:
    print(f"\nTraining with {n_labeled} labeled nodes:")

    # Adjust idx_train to use only the specified number of labeled nodes
    idx_train = idx_train_full[:n_labeled]
```

As mentioned above, I have set the labeled nodes to be considered for GCN training by calling load_data( ) which is defined in utils.py.

Performance metrics considered:

```python
def train(epoch):  1 usage
    t = time.time()
    model.train()
    optimizer.zero_grad()
    output = model(features, adj)
    loss_train = F.nll_loss(output[idx_train], labels[idx_train])
    acc_train = accuracy(output[idx_train], labels[idx_train])
    loss_train.backward()
    optimizer.step()

    if not args.fastmode:
        model.eval()
        output = model(features, adj)

    loss_val = F.nll_loss(output[idx_val], labels[idx_val])
    acc_val = accuracy(output[idx_val], labels[idx_val])

    # Printing only when epoch is a multiple of 5
    if (epoch + 1) % 20 == 0:
        print(
            f'''Epoch: {epoch + 1:04d}, loss_train: {loss_train.item():.4f}, acc_train: {acc_train.item():.4f},
            loss_val: {loss_val.item():.4f}, acc_val: {acc_val.item():.4f}, time: {time.time() - t:.4f}s''')
```

We are considering train accuracy, loss and validation accuracy and loss for each epoch to observe its change in each epoch for further analysis. Also, when I trained the model, the metrics was printed after every epoch, I realized this is hard to document, therefore to reduce the size of output, I have modified the code to print the metrics at epoch == multiples of 20. This way it reduces the output length, and also makes it easier to observe change in performance vs epoch.

In the end for test set:

```python
# Testing function
def test():  1 usage
    model.eval()
    output = model(features, adj)
    loss_test = F.nll_loss(output[idx_test], labels[idx_test])
    acc_test = accuracy(output[idx_test], labels[idx_test])
    print(
        f"Test set results with {n_labeled} labeled nodes: loss= {loss_test.item():.4f}, accuracy= {acc_test.item():.4f}")
```

We calculate test accuracy and loss after training for 200 epochs.

This is how we are executing the code to train for given number of epochs and test the model after training.

Note: I am using the code given in the train.py and utils.py. As mentioned in the question, I have only modified the number of labeled nodes and conditioned print statement to print the metric results only when epoch is multiples of 20.

Performance analysis:

Labeled nodes = 60.

```
Training with 60 labeled nodes:
Epoch: 0020, loss_train: 1.7554, acc_train: 0.3000,
               loss_val: 1.7690, acc_val: 0.3367, time: 0.0364s
Epoch: 0040, loss_train: 1.5084, acc_train: 0.3667,
               loss_val: 1.6479, acc_val: 0.3367, time: 0.0271s
Epoch: 0060, loss_train: 1.2002, acc_train: 0.5667,
               loss_val: 1.4769, acc_val: 0.4967, time: 0.0147s
Epoch: 0080, loss_train: 0.8795, acc_train: 0.8667,
               loss_val: 1.2997, acc_val: 0.6433, time: 0.0102s
Epoch: 0100, loss_train: 0.6819, acc_train: 0.9333,
               loss_val: 1.1421, acc_val: 0.7133, time: 0.0132s
Epoch: 0120, loss_train: 0.5145, acc_train: 0.9167,
               loss_val: 1.0377, acc_val: 0.7267, time: 0.0120s
Epoch: 0140, loss_train: 0.4249, acc_train: 0.9500,
               loss_val: 0.9649, acc_val: 0.7333, time: 0.0125s
Epoch: 0160, loss_train: 0.4195, acc_train: 0.9667,
               loss_val: 0.9175, acc_val: 0.7267, time: 0.0125s
Epoch: 0180, loss_train: 0.3176, acc_train: 0.9667,
               loss_val: 0.8873, acc_val: 0.7233, time: 0.0122s
Epoch: 0200, loss_train: 0.4316, acc_train: 0.9167,
               loss_val: 0.8637, acc_val: 0.7267, time: 0.0153s
Optimization Finished for 60 labeled nodes!
Total time elapsed: 3.2547s
Test set results with 60 labeled nodes: loss= 0.9758, accuracy= 0.6810
```

We can observe that loss and accuracy are inversely proportional to each other. In this case, the maximum training accuracy 0.9167 is at epoch 200 and the minimum training accuracy 0.3000 is at epoch 20. Similarly the validation accuracy is increasing gradually from 0.3367 at epoch 20 to 0.7267 at epoch 200, this suggests the model parameters are improving gradually. But the maximum validation accuracy[0.7267] and test accuracy[0.6810] have significant difference with maximum train accuracy[0.9167]. This suggests at **labeled nodes = 60** that model is showing **over**

fitting characteristics, as it is able to label the data for training data efficiently but failed to do the same with new data[val and test]. This also indicate that model has high variance and increase in labeled nodes could help in reducing this effect.

Labeled nodes = 120.

```
Training with 120 labeled nodes:
Epoch: 0020, loss_train: 1.7944, acc_train: 0.2917,
               loss_val: 1.7785, acc_val: 0.3367, time: 0.0160s
Epoch: 0040, loss_train: 1.5885, acc_train: 0.4333,
               loss_val: 1.6535, acc_val: 0.3633, time: 0.0123s
Epoch: 0060, loss_train: 1.2774, acc_train: 0.6667,
               loss_val: 1.4317, acc_val: 0.6367, time: 0.0102s
Epoch: 0080, loss_train: 1.0012, acc_train: 0.7833,
               loss_val: 1.1932, acc_val: 0.7367, time: 0.0144s
Epoch: 0100, loss_train: 0.8038, acc_train: 0.7917,
               loss_val: 1.0252, acc_val: 0.7500, time: 0.0125s
Epoch: 0120, loss_train: 0.6890, acc_train: 0.8750,
               loss_val: 0.9223, acc_val: 0.7800, time: 0.0157s
Epoch: 0140, loss_train: 0.5748, acc_train: 0.9083,
               loss_val: 0.8494, acc_val: 0.7967, time: 0.0150s
Epoch: 0160, loss_train: 0.5335, acc_train: 0.9167,
               loss_val: 0.7882, acc_val: 0.8133, time: 0.0114s
Epoch: 0180, loss_train: 0.4185, acc_train: 0.9333,
               loss_val: 0.7476, acc_val: 0.8033, time: 0.0089s
Epoch: 0200, loss_train: 0.3949, acc_train: 0.9750,
               loss_val: 0.7188, acc_val: 0.8100, time: 0.0122s
Optimization Finished for 120 labeled nodes!
Total time elapsed: 2.6054s
Test set results with 120 labeled nodes: loss= 0.8734, accuracy= 0.7576
```

In this case, the maximum training accuracy 0.9750 is at epoch 200 and the minimum training accuracy 0.2917 is at epoch 20. Similarly, the validation accuracy is increasing gradually from 0.3367 at epoch 20 to 0.8100 at epoch 200, this suggests the model parameters are improving gradually. But the maximum validation accuracy [0.8100] and test accuracy [0.75100] have significant difference with maximum train accuracy[0.9750]. This suggests that at **labeled nodes = 120** model is showing **over fitting** characteristics, as it is able to label the data for training data efficiently but failed to do the same with new data[val and test]. This also indicates that model has high variance and increase in labeled nodes could help in reducing this effect.

Labeled nodes = 180.

```
Training with 180 labeled nodes:
Epoch: 0020, loss_train: 1.7830, acc_train: 0.2944,
               loss_val: 1.7297, acc_val: 0.3367, time: 0.0116s
Epoch: 0040, loss_train: 1.5207, acc_train: 0.4278,
               loss_val: 1.5629, acc_val: 0.3867, time: 0.0188s
Epoch: 0060, loss_train: 1.2117, acc_train: 0.7111,
               loss_val: 1.2846, acc_val: 0.6833, time: 0.0147s
Epoch: 0080, loss_train: 0.9165, acc_train: 0.7611,
               loss_val: 1.0427, acc_val: 0.7900, time: 0.0102s
Epoch: 0100, loss_train: 0.7627, acc_train: 0.8500,
               loss_val: 0.8852, acc_val: 0.8367, time: 0.0112s
Epoch: 0120, loss_train: 0.6529, acc_train: 0.8778,
               loss_val: 0.7759, acc_val: 0.8333, time: 0.0121s
Epoch: 0140, loss_train: 0.5268, acc_train: 0.9000,
               loss_val: 0.7122, acc_val: 0.8567, time: 0.0125s
Epoch: 0160, loss_train: 0.5347, acc_train: 0.8778,
               loss_val: 0.6642, acc_val: 0.8600, time: 0.0136s
Epoch: 0180, loss_train: 0.4727, acc_train: 0.9333,
               loss_val: 0.6365, acc_val: 0.8533, time: 0.0104s
Epoch: 0200, loss_train: 0.4060, acc_train: 0.9056,
               loss_val: 0.6175, acc_val: 0.8567, time: 0.0149s
Optimization Finished for 180 labeled nodes!
Total time elapsed: 2.3668s
Test set results with 180 labeled nodes: loss= 0.7782, accuracy= 0.7833
```

Compare to previous 2 training results, this has lesser difference between maximum training accuracy [0.9056], maximum validation accuracy [0.8567] and test accuracy [0.7833]. This suggests that with increase in number of labeled nodes, the effect of overfitting characteristics is reducing.

Labeled nodes = 240.

```
Training with 240 labeled nodes:
Epoch: 0020, loss_train: 1.7226, acc_train: 0.3292,
               loss_val: 1.7169, acc_val: 0.3367, time: 0.0118s
Epoch: 0040, loss_train: 1.5154, acc_train: 0.3375,
               loss_val: 1.5314, acc_val: 0.3400, time: 0.0115s
Epoch: 0060, loss_train: 1.2068, acc_train: 0.6292,
               loss_val: 1.2562, acc_val: 0.5833, time: 0.0126s
Epoch: 0080, loss_train: 0.9136, acc_train: 0.8167,
               loss_val: 1.0135, acc_val: 0.7500, time: 0.0111s
Epoch: 0100, loss_train: 0.7523, acc_train: 0.8500,
               loss_val: 0.8572, acc_val: 0.8367, time: 0.0147s
Epoch: 0120, loss_train: 0.6563, acc_train: 0.8583,
               loss_val: 0.7642, acc_val: 0.8567, time: 0.0118s
Epoch: 0140, loss_train: 0.5837, acc_train: 0.8875,
               loss_val: 0.7031, acc_val: 0.8500, time: 0.0133s
Epoch: 0160, loss_train: 0.5564, acc_train: 0.8958,
               loss_val: 0.6631, acc_val: 0.8600, time: 0.0126s
Epoch: 0180, loss_train: 0.5291, acc_train: 0.9042,
               loss_val: 0.6308, acc_val: 0.8600, time: 0.0123s
Epoch: 0200, loss_train: 0.4165, acc_train: 0.9083,
               loss_val: 0.6087, acc_val: 0.8533, time: 0.0115s
Optimization Finished for 240 labeled nodes!
Total time elapsed: 2.4775s
Test set results with 240 labeled nodes: loss= 0.7622, accuracy= 0.8029
```

Similar to previous result, this has lesser difference between maximum training accuracy [0.9083], maximum validation accuracy [0.8533] and test accuracy [0.8029]. This suggests that with increase in number of labeled nodes, the effect of overfitting characteristics is reducing.

Labeled nodes = 300.

```
Training with 300 labeled nodes:
Epoch: 0020, loss_train: 1.7198, acc_train: 0.3200,
               loss_val: 1.7026, acc_val: 0.3367, time: 0.0114s
Epoch: 0040, loss_train: 1.4892, acc_train: 0.3633,
               loss_val: 1.5036, acc_val: 0.3533, time: 0.0114s
Epoch: 0060, loss_train: 1.1678, acc_train: 0.6633,
               loss_val: 1.2144, acc_val: 0.6000, time: 0.0090s
Epoch: 0080, loss_train: 0.9146, acc_train: 0.7733,
               loss_val: 0.9578, acc_val: 0.7933, time: 0.0125s
Epoch: 0100, loss_train: 0.7567, acc_train: 0.8400,
               loss_val: 0.7960, acc_val: 0.8267, time: 0.0207s
Epoch: 0120, loss_train: 0.6311, acc_train: 0.8600,
               loss_val: 0.7067, acc_val: 0.8333, time: 0.0176s
Epoch: 0140, loss_train: 0.5762, acc_train: 0.8733,
               loss_val: 0.6540, acc_val: 0.8367, time: 0.0113s
Epoch: 0160, loss_train: 0.5370, acc_train: 0.8767,
               loss_val: 0.6226, acc_val: 0.8300, time: 0.0173s
Epoch: 0180, loss_train: 0.5006, acc_train: 0.9000,
               loss_val: 0.5934, acc_val: 0.8333, time: 0.0113s
Epoch: 0200, loss_train: 0.4963, acc_train: 0.8867,
               loss_val: 0.5760, acc_val: 0.8400, time: 0.0136s
Optimization Finished for 300 labeled nodes!
Total time elapsed: 2.6212s
Test set results with 300 labeled nodes: loss= 0.7102, accuracy= 0.8129
```

Out of all the above cases, at labeled nodes = 300, we have the least difference between maximum training accuracy [0.8867], validation accuracy [0.8400] and test accuracy [0.8129]. This suggests that with the increase in training set, the effect of overfitting has reduced significantly. So, I can conclude that even for Graph convolutional network, training with small batch of data could end up in overfitting the model, and for deploying the model to label new data, you require some significant size of dataset to train.