A REPORT ON

ACTIVE LEARNING

FOR THE SUBMISSION OF

ASSIGNMENT - 2

BITS F464 - Machine Learning

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

(JUNE 2020)

BY

	TEAM MEMBERS	ID NO.
1.	KANISHK PATIDAR	2018A7PS0228P
2.	AKSHAY GUNDEWAR	2018A7PS0240P
3.	KARTIKAYA SHARMA	2018A7PS0386P

TO

DR. NAVNEET GOYAL

Link to the python notebook having complete code for all parts of the assignment: -

https://colab.research.google.com/drive/1QH10gO_dLNSg-nBesfrjldBkHZxLG6lt#scrollTo=-GptXTVhRmfw

I. Active Learning

Active learning is a special case of supervised learning. In this instead of getting a complete set of labelled data points, a subset of labelled points is used for training. In other words, the model actively selects the valuable data points for training.

II. Dataset

For the training purpose we used two datasets:

1. Optical Recognition of Handwritten Digits Data Set:

This data comprises instances of handwritten numbers from 0 to 9. It is a 10-class dataset with 5620 instances. Every instance represents a vector of 64 dimensions which when treated as positional values on an 8*8 grid which forms the original image of the instance.

https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

2. Wine Data Set

These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. It is a 3-class dataset with 178 instances.

http://archive.ics.uci.edu/ml/datasets/Wine/

III. Supervised learning to Active learning

To convert a supervised learning problem into an active learning problem we can randomly remove data points from the original dataset and train on the rest of labelled points.

```
import numpy as np
from sklearn import datasets

from sklearn.metrics
import classification_report, confusion_matrix,accuracy_score
from sklearn import svm


digits = datasets.load_digits()
rng = np.random.RandomState(1)
indices = np.arange(len(digits.data))
rng.shuffle(indices)
```

```
X = digits.data[indices[:500]]
y = digits.target[indices[:500]]

n_labeled_points = 50

X_train = np.copy(X[:n_labeled_points])
y_train = np.copy(y[:n_labeled_points])

X_test = np.copy(X[50:])
y_test = np.copy(y[50:])
model = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')
model.fit(X_train,y_train)

predicted_labels = model.predict(X_test)

true_labels = y_test

cm = confusion_matrix(true_labels, predicted_labels)
print(cm)

print(accuracy_score(true_labels,
predicted_labels))
```

To train our model we used SVM. We generated the following confusion matrix; rows being 'predicted' and columns being 'actual'. From confusion matrix we can calculate the accuracy of our model, which comes out to be 78%.

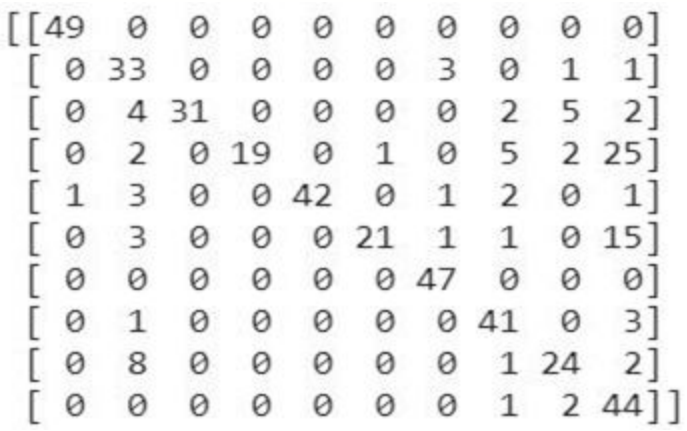

Figure 1: Confusion Matrix

IV. Pool based and Stream based sampling

In pool-based sampling, the machine has access to many examples and samples based on a criterion which judges them on their informativeness. Based on the criteria a user further points are selected to retrain the model for better accuracy.

In stream-based sampling, each unlabelled data point is examined one at a time with the machine evaluating the informativeness of each item against its query parameters. The learner decides for itself whether to assign a label or query the teacher for each datapoint.

To implement pool-based sampling we can run the sampling function incorporated with altered training and testing set in a loop, as we have done

in various instances of our code. To label additional 10% to 40% data points we used the following code snippet.

```
for i in range(4):

 uncertainity_label_indices = np.argsort(uncertainity_labels)

 uncertainity_label_indices = uncertainity_label_indices[:50]


    # sampling function/ technique.


 X_train = np.concatenate((X_train,X_test[uncertainity_label_indices]))

 y_train = np.concatenate((y_train,y_test[uncertainity_label_indice]))

 X_test = np.delete(X_test,uncertainity_label_indices,0)

 y_test = np.delete(y_test,uncertainity_label_indices)
```

V. Uncertainty Sampling

Uncertainty Sampling is a strategy for identifying unlabelled items that are near a decision boundary in the current Machine Learning model. The points lying near the decision boundary of the classifier are the most uncertain points as there is a higher probability of incorrectly predicting the class labels of the corresponding points. Hence these points are labelled in upcoming iterations in active learning giving us better accuracy in upcoming models.

Some of the techniques used in uncertainty sampling are:

1. Least Confident

The simplest measure is the uncertainty of classification defined by,

$$U(x) = 1 - P(\hat{x}|x)$$

where x is the instance to be predicted and x^ is the most likely prediction and P is a probability function. To implement the above formula, we used the following snippet of code.

```
  predicted_prob = model.predict_proba(X_test)
uncertainity_labels = 1 - predicted_prob.max(axis=1)
```

2. Margin Sampling

Classification margin is the difference in probability of the first and second most likely prediction, that is, it is defined by,

$$M(x) = P(\hat{x_1}|x) - P(\hat{x_2}|x)$$

where x1^ and x2^ are the first and second most likely classes. To implement the above formula, we used the following snippet of code.

```
  predicted_prob = model.predict_proba(X_test)
part = np.partition(-predicted_prob, 1, axis=1)
uncertainity_labels = - part[:, 0] + part[:, 1]
```

3. Entropy

The third built-in uncertainty measure is the classification entropy, which is defined by,

$$H(x) = -\sum_{k} p_k \log(p_k)$$

where pk is the probability of the sample belonging to the k-th class. To implement the above formula, we used the following snippet of code.

```
predicted_prob = model.predict_proba(X_test)

uncertainity_labels = entropy(predicted_prob.T)
```

We calculated probabilities for each iteration for three techniques mentioned above. We got the following results:

Iteration	Least Confident	Margin Sampling	Entropy
initial(10%)	0.865	0.865	0.865
Additional(10%)	0.868	0.934	0.905
Additional(20%)	0.862	0.993	0.908
Additional(30%)	0.851	0.997	0.912
Additional(40%)	0.89	0.996	0.922

Figure 2: Accuracy Distribution

VI. Query by Committee

Query by committee is an active learning strategy which alleviates many disadvantages of uncertainty sampling. In this method a committee of different models of classification are formed. Every committee gets to vote for the class label of every point in the dataset A query is generated for each point for every committee member. To decide the share of vote disagreement sampling is used. More the merrier, points with the most disagreement are the most informative ones.

To form a committee, we used bagging (bootstrapping algorithm). This algorithm forms n subsets of a dataset where n is the number of committee members. Following snippet portrays the above-mentioned strategy.

```
r1 = np.random.RandomState(1)

r2 = np.random.RandomState(2)

indices1_QBC = np.arange(len(X_train))

indices2_QBC = np.arange(len(X_test))


X_QBC_train = np.random.rand(5,50,64)

y_QBC_train = np.random.rand(5,50)

X_QBC_test = np.random.rand(5,450,64)

y_QBC_test = np.random.rand(5,450)
```

```
for i in range(5):

    r1.shuffle(indices1_QBC)

    r2.shuffle(indices2_QBC)

    X_QBC_train[i] = X_train[indices1_QBC[:50]]

    y_QBC_train[i] = y_train[indices1_QBC[:50]]

    X_QBC_test[i] = X_test[indices2_QBC[:450]]

    y_QBC_test[i] = y_test[indices2_QBC[:450]]
```

Using these new datasets, we created 5 new models in every iteration of the pool-based scenario.

```
for _ in range(5):

    c1 = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')

    c1.fit(X_QBC_train[0],y_QBC_train[0])

    c2 = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')

    c2.fit(X_QBC_train[1],y_QBC_train[1])

    c3 = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')

    c3.fit(X_QBC_train[2],y_QBC_train[2])

    c4 = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')

    c4.fit(X_QBC_train[3],y_QBC_train[3])

    c5 = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')
```

```
    c5.fit(X_QBC_train[4],y_QBC_train[4])

print(c1.score(X_test,y_test),c2.score(X_test,y_test),c3.score(X_test,y_te
st),c4.score(X_test,y_test),c5.score(X_test,y_test))

    p[:,0] = c1.predict(X_test)

    p[:,1] = c2.predict(X_test)

    p[:,2] = c3.predict(X_test)

    p[:,3] = c4.predict(X_test)

    p[:,4] = c5.predict(X_test)
```

To decide the effective contribution of each committee member disagreement sampling is used. Some examples of disagreement sampling are:

1. Vote entropy:

In the above formula V is the vote function, C is the number of committees. To implement the above-mentioned formula:

```
    Vote = np.arange(10*900).reshape(900,10)

    Vote[:,:] = 0

    for i in range(len(y_test)):

        for j in range(5):

            Vote[i,p[i,j]-1]+=1

    Prob = np.divide(Vote,5)

    uncertainity_labels = entropy(Prob.T)

    uncertainity_label_indices = np.argsort(uncertainity_labels)

    uncertainity_label_indices = uncertainity_label_indices[:100]
```

Iteration	C1	C2	C3	C4	C5	Commit-tee
initial(10%)	0.8411	0.6544	0.6889	0.6733	0.6611	0.899
Additional(10%)	0.62	0.5844	0.6789	0.6322	0.6167	0.94
Additional(20%)	0.6623	0.6211	0.6122	0.6478	0.6178	0.947
Additional(30%)	0.66667	0.6533	0.6289	0.6222	0.7089	0.959
Additional(40%)	0.6589	0.7422	0.6911	0.6623	0.6289	0.96

Figure 3: Accuracy Distribution for QBC

2. KL Divergence:

To implement the above formula, we assigned p as probability function and q as mean of the sum of probabilities for every instance which comes out to be 0.1 (1/no. of classes).

```
  Vote = np.arange(10*900).reshape(900,10)
Vote[:,:] = 0
for i in range(len(y_test)):
    for j in range(5):
        Vote[i,p[i,j]-1]+=1

Prob = np.divide(Vote,5)
```

```
uncertainity_labels = entropy(Prob.T)

uncertainity_label_indices = np.argsort(uncertainity_labels)

uncertainity_label_indices = uncertainity_label_indices[:100]
```

After every iteration we altered the training data as per the following code:

```
X_train = np.concatenate((X_train,X_test[uncertainity_label_indices]))

y_train = np.concatenate((y_train,y_test[uncertainity_label_indices]))


for i in range(5):

    r1 = np.random.RandomState(i)

    r2 = np.random.RandomState(i*i)


    r1.shuffle(indices1_QBC)

    r2.shuffle(indices2_QBC)


    X_QBC_train[i] = X_train[indices1_QBC[:50]]

    y_QBC_train[i] = y_train[indices1_QBC[:50]]

    X_QBC_test[i] = X_test[indices2_QBC[:450]]

    y_QBC_test[i] = y_test[indices2_QBC[:450]]
```

Iteration	C1	C2	C3	C4	C5	Commit-tee
initial(10%)	0.8411	0.6544	0.6889	0.6733	0.6611	0.934
Additional(10%)	0.62	0.5844	0.6789	0.6322	0.6167	0.964
Additional(20%)	0.6623	0.6211	0.6122	0.6478	0.6178	0.969
Additional(30%)	0.6667	0.6533	0.6289	0.6222	0.7089	0.968
Additional(40%)	0.6589	0.7422	0.6911	0.6623	0.6289	0.971

Figure 4: Accuracy Distribution for QBC

Iteration	Vote Entropy	KL Divergence
initial(10%)	0.899	0.934
Additional(10%)	0.94	0.964
Additional(20%)	0.947	0.969
Additional(30%)	0.959	0.968
Additional(40%)	0.96	0.971

Figure 5: Accuracy Comparison

VII. Version Space

Version space is the set of hypotheses that are consistent with the current labeled training data. In other words, version space is the region between the

outermost boundary formed by multiple classifiers, containing the boundary of every classifier of the models.

To get points comprising version space we can find the points that have disagreement among at least 2 classifiers. The points with the maximum no. of unique votes are the one that, on labelling, reduces the version space by maximum. We implemented the above strategy using the following code.

```
for i in range(len(Vote)):

    if(Vote[i,np.argmax(Vote[i])]!=5):

        version_indices.append(i)

print("The size/number of points in of verson space is: " +
str(len(version_indices)))

version_vote = Vote[version_indices]



version_unique_count = []

for i in range(len(version_vote)):

    version_unique_count.append(len(np.unique(version_vote[i])))

version_unique_count = np.multiply(version_unique_count,-1)

version_sort_indices = np.argsort(version_unique_count)
```

Upon executing the above code, we found that the size of version space is 423 and we will remove the points from the test and add it to train as per the entries of version_sort_indices.

VIII. Selection of labelling points randomly

```
X = digits.data[indices[:1000]]

y = digits.target[indices[:1000]]

images = digits.images[indices[:1000]]


n_total_samples = len(y)

n_labeled_points = 100


model = svm.SVC(decision_function_shape='ovo',probability=True,C=1.0,
kernel='rbf')


for i in range(5):

    X_train = X[:50*(i+1)]

    y_train = y[:50*(i+1)]


    X_test = X[50*(i+1):]

    y_test = y[50*(i+1):]


    model.fit(X_train,y_train)

    predicted_labels = model.predict(X)

    true_labels = y

    print(accuracy_score(true_labels, predicted_labels))
```

Using stream-based scenario:

In a stream-based scenario,we used the base model to find the informativeness of an unlabelled data point. Thresholding can be done by hit and trial keeping in mind the percentage of points you need to label. So, we sorted the data according to their probabilities and chose the required(50) number of points.

```
predicted_prob = model.predict_proba(X_test)

a = []

for i in range(len(predicted_prob)):

    a.append(predicted_prob[i][np.argmax(predicted_prob[i])])

a = np.array(a)

ind = np.argsort(a,axis=0)



for i in range(4):

  X_train = np.concatenate((X_train,X_test[ind[400-i*50:]]))

  y_train = np.concatenate((y_train,y_test[ind[400-i*50:]]))



  model.fit(X_train,y_train)



  predicted_labels = model.predict(X)

  true_labels = y

  print(accuracy_score(true_labels, predicted_labels))
```

Iteration	Vote Entropy	KL Divergence	Random points	Stream based approach
initial(10%)	0.899	0.934	0.776	0.8
Additional(10%)	0.94	0.964	0.865	0.804
Additional(20%)	0.947	0.969	0.925	0.81
Additional(30%)	0.959	0.968	0.943	0.82
Additional(40%)	0.96	0.971	0.957	0.842

Figure 6: Accuracy Comparison

IX. K-means

We take the data set, then select randomly k data points and assign then as means. Now we assign every data point a cluster based on the distance of that data point from the means. A data point is assigned to the cluster whose mean is nearest to it.

Now we update the means by calculating the centroids for each cluster. Then we repeat the above steps till the point where the means remain the same. This point is called convergence.

Now we label some random points from each cluster and get the label which is mode for the labels. The whole cluster is assigned the label which has the highest frequency.

Now we compare the new labeling with the true data and calculate the accuracies.

Importing the required libraries

```
import numpy as np

import math

import statistics

from scipy import stats

from sklearn import datasets

from sklearn.metrics import classification_report,
confusion_matrix,accuracy_score
```

Loading the dataset and randomizing them

```
digits = datasets.load_digits()

rng = np.random.RandomState(1)

indices = np.arange(len(digits.data))

rng.shuffle(indices)



X = digits.data[indices[:500]]

y = digits.target[indices[:500]]

images = digits.images[indices[:500]]
```

```
x_labeled_10 = np.copy(X[:50])

y_labeled_10 = np.copy(y[:50])


x_unlabeled_90 = np.copy(X[50:])

y_unlabeled_90 = np.copy(y[50:])
```

Selecting the 40% from the unlabeled data points for further clustering

```
x_random_40 = np.copy(X_unlabeled_90[:200])

y_random_40 = np.copy(y_unlabeled_90[:200])
```

Initialising the random seeds for clustering

```
k_means = x_random_40[:10]

k_means_cur = x_random_40[20:30]
```

Initialising the variables to store the final results

```
money = 0

time = 0

clusters = np.zeros(200,dtype = int)

y_final = np.zeros(200,dtype = int)
```

```
distances = np.zeros(10)

flag = False
```

Starting the loop for clustering

```
for i in range(10000):

  for j in range(200):

    for k in range(10):
```

Calculating the distance of every data point from k_means and assigning the cluster of the mean that is nearest to the data point.

```
      distances[k] = np.linalg.norm(x_random_40[j] - k_means[k])

    c = np.argmin(distances)

    clusters[j] = c
```

Assigning the new means by calculating the centroids.

```
  for j in range(10):

    c = np.where(clusters == j)

    s = np.average(x_random_40[c], axis = 0)

    k_means_cur[j] = s
```

Checking if the model has converged that is the point when the clustering doesn't change anymore.

```
  flag = np.array_equal(k_means_cur,k_means)
  if(flag):
    break
  k_means = np.copy(k_means_cur)
```

Getting random 20% from each cluster and labeling them and updating the final classification (y_final) by assigning the whole cluster with the label that is the mode in the labeled data.

```
for j in range(10):
    c = np.where(clusters == j)
    len_1 = int(0.2*len(c[0])+1)
    labeled = y_random_40[c[0][:len_1]]
    money += len(labeled) * 100
    time += len(labeled)

    try:
      most_freq = statistics.mode(labeled)
```

```
    except statistics.StatisticsError:
      most_freq = labeled[0]
    y_final[c] = most_freq
```

Finally printing the outputs. The confusion matrix, accuracy and cost for all the labeling

```
cm = confusion_matrix(y_final, y_random_40)
print(cm)
print("the accuracy is : ",accuracy_score(y_final, y_random_40))
print("it would take ", money, "rupees \nand it would take ", time, "hours to label")
```

Output :

```
[[21  1  1  0  2  0  0  0  0  0]
 [ 0  4  0  0  2  0  0  1  0  3]
 [ 0 11 16  1  0  0  0  6  4  0]
 [ 0  0  6 22  0  0  0  0  0  0]
 [ 0  0  0  1 13  0  0  0  0  0]
 [ 0  0  0  1  0 17  0  0  2  0]
 [ 0  0  0  0  0  0 21  0  0  1]
 [ 0  1  0  1  2  0  0 13  2  1]
 [ 0  0  0  0  0  0  0  0  0  0]
 [ 0  0  1  5  0  0  0  0  6 11]]
the accuracy is :  0.69
it would take  4600 rupees
and it would take  46 hours to label
```

Figure 7: Confusion Matrix

X. Self Organizing Maps (SOMs)

Self Organizing Maps (SOMs) are used to represent a high dimensional data into a visualisable 2D feature space. The geometric relationship between points is used to indicate similarity between the points. The nearer points would have more in common than the points which are far away from them. It is an unsupervised dimensionality reduction and visualisation algorithm which allows us to study data which otherwise would be undecipherable.

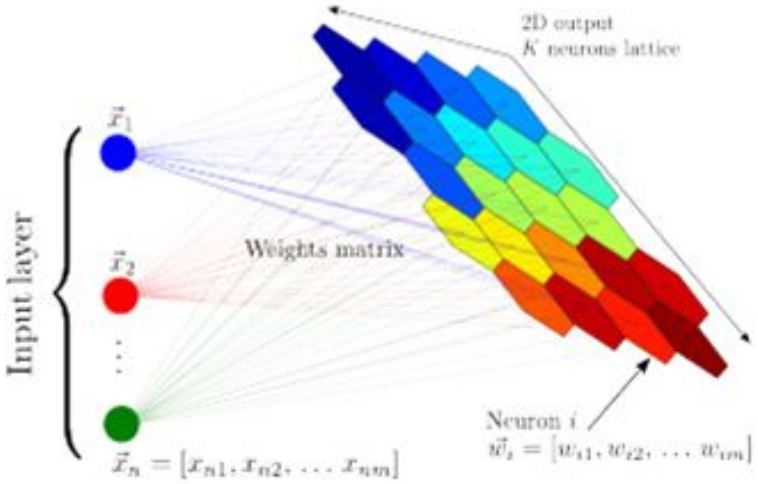

Figure 8: SOM Architecture

To implement this algorithm, we create a 2D hexagonal grid of required shape in which each hexagon represents a neuron initialised with random weights. The dimensions of the weight vector are the same as the dimensions of the input vector. After the weights are initialized, we select a sample from input space one-by-one. For each input vector, we select a “winning” neuron, which has the least Euclidean distance to the input vector. Since, this neuron closely resembles the input vector, so we select it as the winning one. After that, we will update the weights of adjacent neurons so that the

closer neurons are more similar to it than the farther ones. The shape of neighbourhood can be any like square, hexagonal, gaussian etc.

In this question, we have used the Wine dataset which is mentioned above to train our SOM network. Also, the dimensions of output hexagonal grid used is 10 by 10. The neighbourhood used by default is “Gaussian”

First, we will import the required libraries and frameworks.

```
import pandas as pd

import SimpSOM as sps

import numpy as np

from sklearn.datasets import load_wine
```

Now, we will load the wine dataset: -

```
dataset = load_wine()

x = dataset.data

y = dataset.target
```

So, once we have loaded the dataset now we can train the SOM network: -

```
net = sps.somNet(10, 10, x, PBC=True)

net.train(0.01, 20000)
```

This makes our initial SOM (10 by 10) network and saves it in the variable named “net”. Then this net is trained with a learning rate of alpha = 0.01 and for 20 thousand epochs over the wine dataset.

So, the weight differences can be visualized as seen in the below figure:

```
net.diff_graph()
```

SOMs are also used to visualize the effect of individual attributes on the whole training set.

As mentioned earlier, each wine tuple has 13 attributes as mentioned below:

1)Alcohol

2) Malic acid

3) Ash

4) Alkalinity of ash

5) Magnesium

6) Total phenols

7) Flavanoids

8) Nonflavanoid phenols

9) Proanthocyanins

10)Color intensity

11)Hue

12)OD280/OD315 of diluted wines

13)Proline

Since this is a 13D data, we can now visualize the data for each individual attribute. Some of which are shown below:

```
net.nodes_graph(colnum=0)
```

Fig.: SOM denoting the alcohol content

```
net.nodes_graph(colnum=3)
```

Fig.: SOM denoting the alkalinity of ash

```
net.nodes_graph(colnum=9)
```

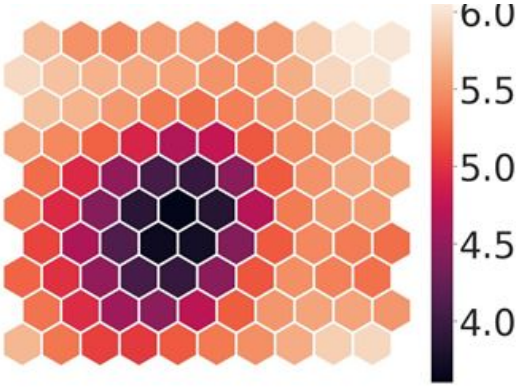

Fig.: SOM denoting the colour intensity of wine