federated_models_clustering_k_means

June 19, 2020

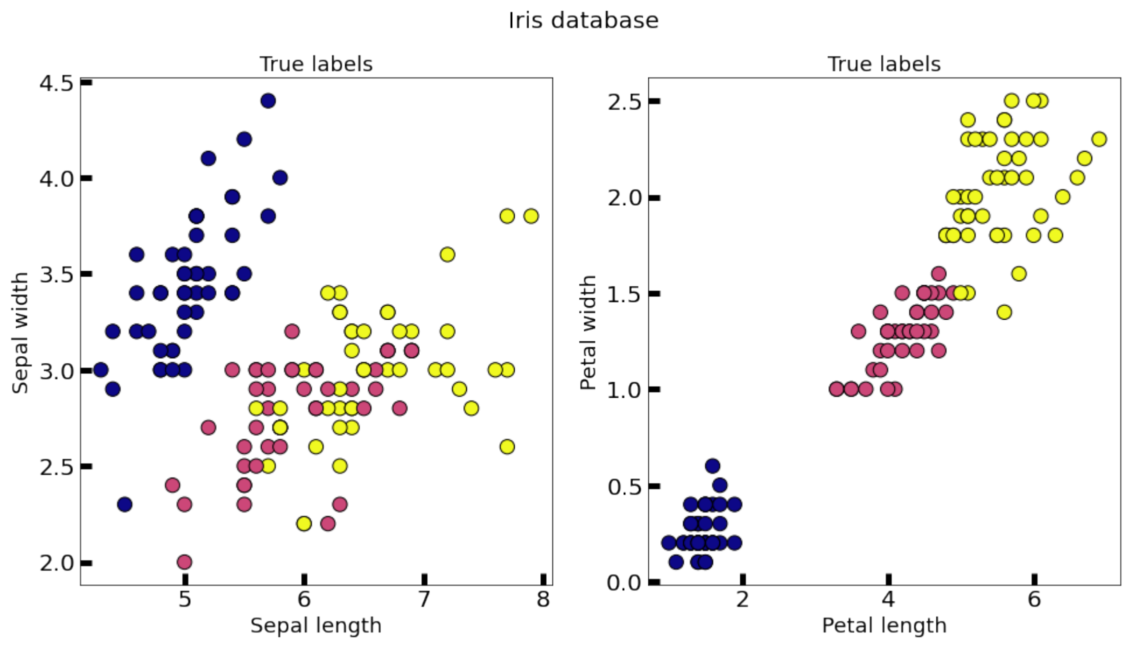
1 Unsupervised federated learning: K-means clustering

The present notebook tackles the problem of *unsupervised* learning in a federated configuration. In particular, a K-Means clustering is used from the `sklearn` library (see this link).

The framework provides some functions to load the Iris dataset.

[1]:
```
import matplotlib.pyplot as plt
import shfl
import numpy as np
from shfl.data_base.iris import Iris


# Assign database:
database = Iris()

train_data, train_labels, test_data, test_labels = database.load_data()
print(train_data.shape)
print(test_data.shape)

# Visualize train data:
fig, ax = plt.subplots(1,2, figsize=(16,8))
fig.suptitle("Iris database", fontsize=20)
ax[0].set_title('True labels', fontsize=18)
ax[0].scatter(train_data[:, 0], train_data[:, 1], c=train_labels, s=150,␣
 ↪edgecolor='k', cmap="plasma")
ax[0].set_xlabel('Sepal length', fontsize=18)
ax[0].set_ylabel('Sepal width', fontsize=18)
ax[0].tick_params(direction='in', length=10, width=5, colors='k', labelsize=20)

ax[1].set_title('True labels', fontsize=18)
ax[1].scatter(train_data[:, 2], train_data[:, 3], c=train_labels, s=150,␣
 ↪edgecolor='k', cmap="plasma")
ax[1].set_xlabel('Petal length', fontsize=18)
ax[1].set_ylabel('Petal width', fontsize=18)
ax[1].tick_params(direction='in', length=10, width=5, colors='k', labelsize=20)

plt.show()
```

```
(135, 4)
(15, 4)
```



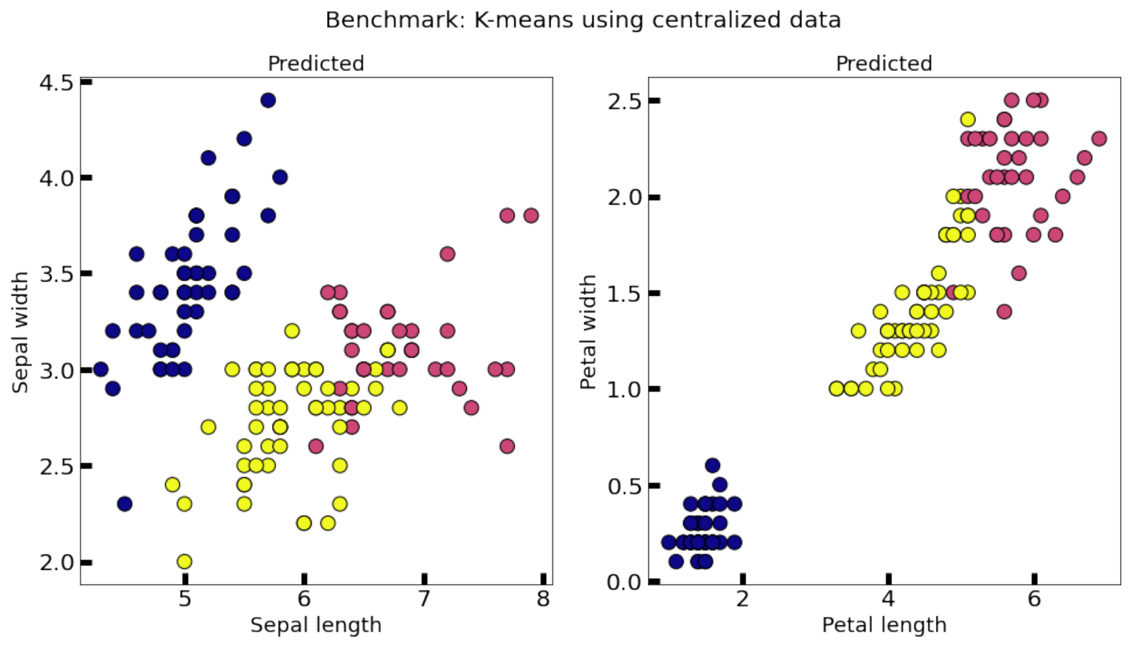
We implement a method to plot K-Means results in Iris database and establish a centralised model which will be our reference model.

[2]:
```
from shfl.model.kmeans_model import KMeansModel

def plot_k_means(km, X, title):
    new_labels = km.predict(X)
    fig, axes = plt.subplots(1, 2, figsize=(16,8))
    fig.suptitle(title, fontsize=20)
    axes[0].scatter(X[:, 0], X[:, 1], c=new_labels, cmap='plasma',␣
 ↪edgecolor='k', s=150)
    axes[0].set_xlabel('Sepal length', fontsize=18)
    axes[0].set_ylabel('Sepal width', fontsize=18)
    axes[0].tick_params(direction='in', length=10, width=5, colors='k',␣
 ↪labelsize=20)
    axes[0].set_title('Predicted', fontsize=18)

    axes[1].scatter(X[:, 2], X[:, 3], c=new_labels, cmap='plasma',␣
 ↪edgecolor='k', s=150)
    axes[1].set_xlabel('Petal length', fontsize=18)
    axes[1].set_ylabel('Petal width', fontsize=18)
    axes[1].tick_params(direction='in', length=10, width=5, colors='k',␣
 ↪labelsize=20)
    axes[1].set_title('Predicted', fontsize=18)
```

```python
# Plot train data:
centralized_model = KMeansModel(n_clusters=3, n_features = train_data.shape[1],␣
 ↪init = np.zeros((3,4)))
centralized_model.train(train_data)

print(centralized_model.get_model_params())
plot_k_means(centralized_model, train_data, title = "Benchmark: K-means using␣
 ↪centralized data")
```

```
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
```

```
[[5.02826087 3.44347826 1.46086957 0.25      ]
 [6.83428571 3.1        5.73428571 2.09142857]
 [5.9037037  2.73888889 4.38703704 1.43148148]]
```



How to aggregate model's parameters from each federated node in clustering

Since the labels of clusters can vary among each node, we cannot average the centroids right away. A solution is to choose the lowest distance average: this is achieved by simply applying the k-means algorithm on the centroids coordinates of all nodes. In ClusterFedAvgAggregator you can see its implementation.

Remark: this implementation is based on the assumption that the number of clusters is previously fixed across the clients, so it only works properly in I.I.D scenarios (see Federated Sampling). We are working in a federated aggregation operator which works in every distribution of data across clients.

[3]:
```
from shfl.federated_aggregator.cluster_fedavg_aggregator import␣
↪ClusterFedAvgAggregator

# Create the IID data:
iid_distribution = shfl.data_distribution.IidDataDistribution(database)
federated_data, test_data, test_label = iid_distribution.
↪get_federated_data(num_nodes = 12, percent=100)
print("Number of nodes: " + str(federated_data.num_nodes()))

# Run the algorithm:
aggregator = ClusterFedAvgAggregator()
```

```
Number of nodes: 12
```

We are now ready to run our model in a federated configuration.

The performance is assessed by several clustering metrics (see this link).

For reference, below we compare the metrics of: - Each node; - The global (federated) model; - The centralized (non-federated) model.

It can be observed that the performance of *Global federated model* is in general superior with respect to the performance of each node, thus the federated learning approach proves to be beneficial. Moreover, the performance of the Global federated model is very close to the performance of the centralized model.

[4]:
```
from shfl.federated_government.federated_government import FederatedGovernment

n_clusters = 3 # Set number of clusters
n_features = train_data.shape[1]
def model_builder():
    model = KMeansModel(n_clusters=n_clusters, n_features = n_features)
    return model


federated_government = FederatedGovernment(model_builder, federated_data,␣
↪aggregator)
print("Test data size: " + str(test_data.shape[0]))
print("\n")
federated_government.run_rounds(n = 3, test_data = test_data, test_label =␣
↪test_label)

# Reference Centralized (non federate) model:
print("Centralized model test performance : " + str(centralized_model.
↪evaluate(data=test_data, labels=test_labels)))
plot_k_means(centralized_model, test_data, title = "Benchmark on Test data:␣
↪K-means using CENTRALIZED data")
```

```
plot_k_means(federated_government.global_model, test_data, title = "Benchmark␣
↪on Test data: K-means using FL")
```

```
Test data size: 15


Accuracy round 0
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2f10>: (0.5749379307050223, 0.5472979980462193,
0.5607775876964347, 0.3581907090464548)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2090>: (0.6748787177226521, 0.6748787177226521,
0.6748787177226521, 0.6244038155802861)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2b90>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eacc10>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eace10>: (0.5768531209780225, 0.5768531209780225,
0.5768531209780225, 0.41573926868044514)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6250>: (0.5966349856670687, 0.7498564175054125,
0.664527930674641, 0.4621295279912184)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6950>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6190>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6c50>: (0.7926937347958167, 0.862763632213387,
0.8262457729138631, 0.7987734764277501)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6410>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6b50>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6590>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Global model test performance : (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
```

```
Accuracy round 1
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2f10>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2090>: (0.6748787177226523, 0.6748787177226523,
0.6748787177226523, 0.6244038155802861)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2b90>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eacc10>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eace10>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6250>: (0.4731999883759308, 0.44024532963503965,
0.4561282011264447, 0.30664465538588526)
```

```
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
```

```
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
```

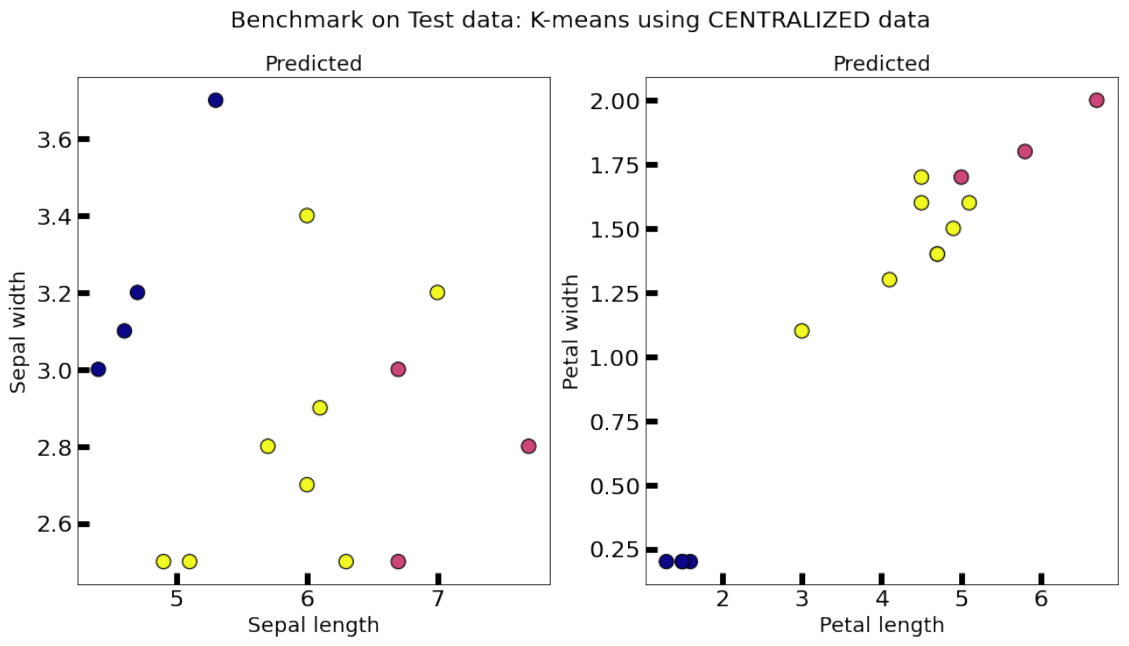
```
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
```

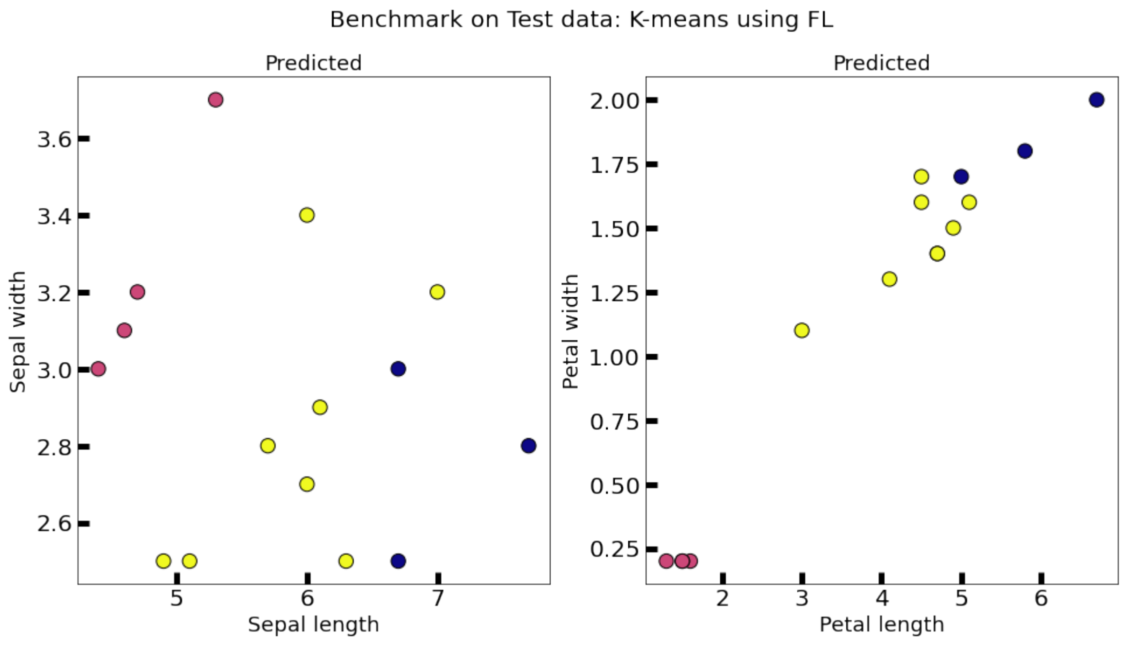
```
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6950>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6190>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6c50>: (0.7926937347958166, 0.862763632213387,
0.826245772913863, 0.7987734764277501)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6410>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6b50>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6590>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Global model test performance : (0.6748787177226521, 0.6748787177226521,
0.6748787177226521, 0.6244038155802861)



Accuracy round 2
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2f10>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2090>: (0.6748787177226521, 0.6748787177226521,
0.6748787177226521, 0.6244038155802861)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2b90>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eacc10>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eace10>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6250>: (0.4731999883759308, 0.44024532963503965,
0.4561282011264447, 0.30664465538588526)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6950>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6190>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6c50>: (0.7926937347958167, 0.862763632213387,
0.8262457729138631, 0.7987734764277501)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6410>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
```

```
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
/usr/local/lib/python3.7/site-packages/shfl/model/kmeans_model.py:38:
RuntimeWarning: Explicit initial center position passed: performing only one
init in k-means instead of n_init=10
  self._k_means.fit(data)
```

```
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6b50>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6590>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Global model test performance : (0.6748787177226521, 0.6748787177226521,
0.6748787177226521, 0.6244038155802861)
```

```
Centralized model test performance : (0.6748787177226522, 0.6748787177226522,
0.6748787177226522, 0.6244038155802861)
```





Differentially Private version

To preserve the privacy of the clients, in this section we introduce Differential Privacy (DP) in our model. Firstly we calibrate the noise introduced by the differentially private mechanism using the train data, then we apply DP to each client feature, so that each cluster computed by a client is shared with the main server privately, that is, without disclosing client's identity.

In the case of applying the Gaussian privacy mechanism , the noise added has to be of the order of the sensitivity of the model's output, i.e. the coordinates of each cluster.

In the general case, the model's sensitivity might be difficult to compute analytically. An alternative approach is to attain *random* differential privacy through a sampling over the data.

That is, instead of computing analytically the *global* sensitivity Δf, we compute an *empirical estimation* of it by sampling over the dataset. This approach is very convenient since allows for the sensitivity estimation of an arbitrary model or a black-box computer function. The `Sherpa.FL` framework provides this functionality in the class `SensitivitySampler`.

In order to carry out this approach, we need to specify a distribution of the data to sample from. This in general requires previous knowledge and/or model assumptions. However, in our specific case of manufactured data, we may assume that the data distribution is *uniform*. To the end, we define our class of `ProbabilityDistribution` that uniformly samples over a data-frame. Moreover, we assume that we do have access to a set of data (this can be thought, for example, as some reference public data set). In this example, we generate a *new* dataset, and use its train partition for sampling:

```
[5]: import numpy as np

     class UniformDistribution(shfl.differential_privacy.ProbabilityDistribution):
         """
         Implement Uniform sampling over the data
         """
         def __init__(self, sample_data):
             self._sample_data = sample_data

         def sample(self, sample_size):
             row_indices = np.random.randint(low=0, high=self._sample_data.shape[0],␣
      ↪size=sample_size, dtype='l')

             return self._sample_data[row_indices, :]

     sample_data = train_data
```

The class `SensitivitySampler` implements the sampling given a *query*, i.e. the learning model itself in this case. We only need to add the method `get` to our model since it is required by the class `SensitivitySampler`. We choose the sensitivity norm to be the v norm and we apply the sampling. The value of the sensitivity depends on the number of samples `n`: the more samples we perform, the more accurate the sensitivity. Indeed, increasing the number of samples `n`, the sensitivity gets more accurate and typically decreases.

Unfortunately, sampling over a dataset involves, at each sample, the training of the model on two datasets differing in one entry. Thus in general this procedure might be computationally expensive (e.g. in the case of training a deep neuronal network).

```
[6]: from shfl.differential_privacy import SensitivitySampler
     from shfl.differential_privacy import L2SensitivityNorm
```

```
class KMeansSample(KMeansModel):

    def __init__(self, feature, **kargs):
        self._feature = feature
        super().__init__(**kargs)

    def get(self, data_array):
        self.train(data_array)
        params = self.get_model_params()
        return params[:, self._feature]

distribution = UniformDistribution(sample_data)
sampler = SensitivitySampler()
# Reproducibility
np.random.seed(789)
n_samples = 50

sensitivities = np.empty(n_features)

for i in range(n_features):
    model = KMeansSample(feature=i, n_clusters=n_clusters,␣
 ↪n_features=n_features)
    sensitivities[i], _ = sampler.sample_sensitivity(model,␣
 ↪L2SensitivityNorm(), distribution, n=n_samples, gamma=0.05)
```

```
[7]: print("Max sensitivity from sampling: ", np.max(sensitivities))
print("Min sensitivity from sampling: ", np.min(sensitivities))
print("Mean sensitivity from sampling:", np.mean(sensitivities))
```

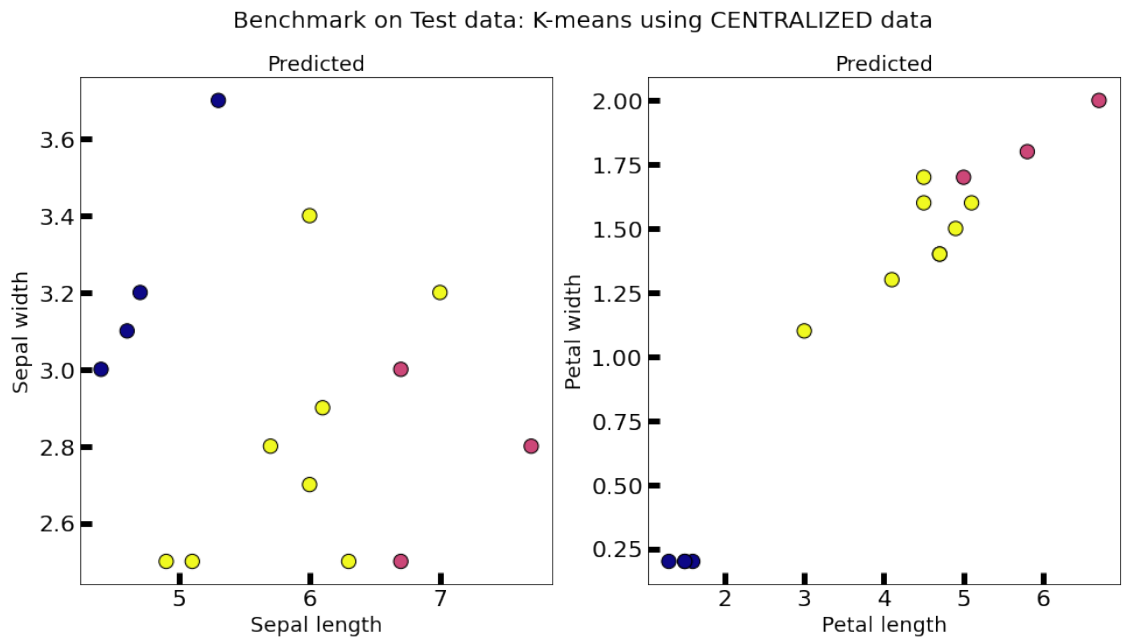
```
Max sensitivity from sampling:  6.571840281064531
Min sensitivity from sampling:  1.6137680902098643
Mean sensitivity from sampling: 3.5674446474034647
```

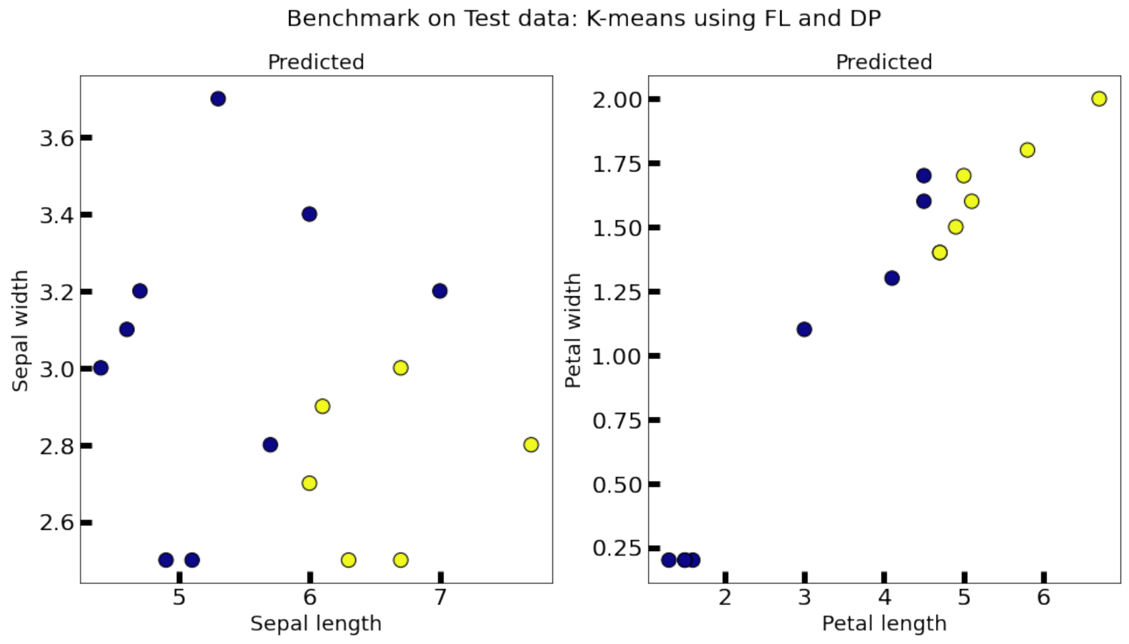
Generally if the model has more than one feature, it is a bad idea to estimate the sensitivity for all of the features at the same time as the features may have wildly varying sensitivities. In this case we estimate the sensitivity for each feature. Note that we provide the array of estimated sensitivities to the GaussianMechanism and it applies it to each feature individually.

```
[8]: from shfl.differential_privacy import GaussianMechanism

dpm = GaussianMechanism(sensitivity=sensitivities, epsilon_delta=(0.9, 0.9))
federated_government = FederatedGovernment(
    model_builder, federated_data, aggregator, model_params_access = dpm)
print("Test data size: " + str(test_data.shape[0]))
print("\n")
federated_government.run_rounds(n = 1, test_data = test_data, test_label =␣
 ↪test_label)
```

```
# Reference Centralized (non federate) model:
print("Centralized model test performance : " + str(centralized_model.
 ↪evaluate(data=test_data, labels=test_labels)))
plot_k_means(centralized_model, test_data, title = "Benchmark on Test data:␣
 ↪K-means using CENTRALIZED data")
plot_k_means(federated_government.global_model, test_data, title = "Benchmark␣
 ↪on Test data: K-means using FL and DP")
```

```
Test data size: 15


Accuracy round 0
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2f10>: (0.5749379307050224, 0.5472979980462194,
0.5607775876964349, 0.3581907090464548)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2090>: (0.6748787177226521, 0.6748787177226521,
0.6748787177226521, 0.6244038155802861)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ea2b90>: (0.6273556493217654, 0.5971957538368565,
0.611904292386804, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eacc10>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135eace10>: (0.5768531209780225, 0.5768531209780225,
0.5768531209780225, 0.41573926868044514)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6250>: (0.5966349856670687, 0.7498564175054125,
0.664527930674641, 0.4621295279912184)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6950>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6190>: (0.6357568516948604, 0.6616415018902373,
0.648440961914333, 0.5877502944640753)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6c50>: (0.7926937347958167, 0.862763632213387,
0.8262457729138631, 0.7987734764277501)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6410>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6b50>: (0.6273556493217655, 0.5971957538368566,
0.6119042923868042, 0.4865525672371638)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x135ed6590>: (0.599290617221163, 0.5575547375460104,
0.5776698180471677, 0.3933140734626496)
Global model test performance : (0.17435413852721285, 0.2615561448346113,
0.20923294563016676, 0.03951701427003293)


Centralized model test performance : (0.6748787177226522, 0.6748787177226522,
0.6748787177226522, 0.6244038155802861)
```





As you can see when we add DP to the model it becomes quite unstable (multiple executions each one with very different results) and almost useless (even with unacceptable values for δ, that is $\delta \geq 0.5$, the results are quite bad), which suggest that another way of adding DP have to be provided. An alternative approach for adding DP can be found in A differential privacy protecting K-means clustering algorithm based on contour coefficients, but still it is unclear how to adapt it in a federated setting.

[ ]:

```
```