Queen Mary University of London

Department of Electrical Engineering & Computer Science

ECS766P Data Mining – Assignment 2

Hasan Emre Erdemoglu

10 November 2020

Table of Contents:

Any code within this report could be provided upon request.

Part 1: Oct 20, 2020

Question 1.1:

Part A:

With the information given in the question the following star scheme can be drawn. Specified measures are also added below to fact tables and dimension tables as well. Given more information this schema can be expanded or transformed to snowflake or constellation schemas if required.

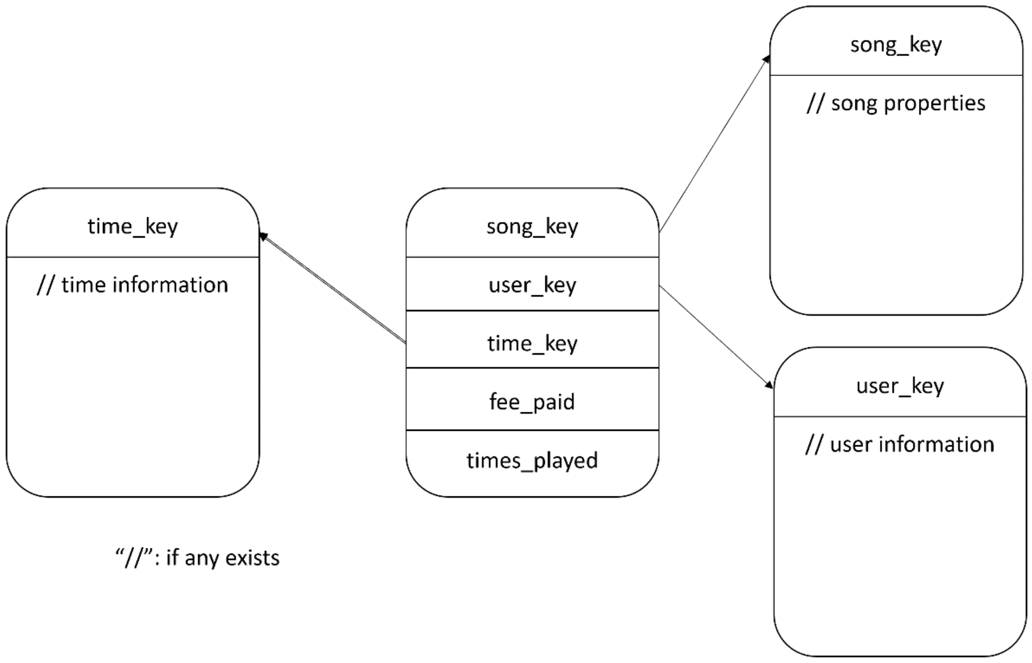


Figure 1. Star Schema Representation for Question 1.1

Part B:

Starting from the base cuboid (song, user, time) to fetch total fee collected for a given song the following OLAP operations must be executed in order:

1. On time dimension, slice the year of interest.
2. On time dimension; drill down from year to month.
3. Again, on time dimension, slice the month of interest. This will give us the User and Song dimensions for that month and the year.
4. Slice the song of interest. This will output the number of times that all users played that song at that given month and year and also the fee that has to be paid for that particular song. (times_played)
5. To calculate the total fee collected; sum over user times_played values and multiply by fee_paid.

Part C:

The equations needed for this part can be found at Slide 38 of Week 5.

Without the hierarchy as in the previous section; the number of cuboids would be 2^N where N is the number of dimensions available. With hierarchies the following formula hold for computing the number of cuboids:

$$T = \prod_{i=1}^{n} (L_i + 1)$$

Using the formula, we have 6 {time} * 2 {song} * 2 {use} = 24 cuboids.

Question 1.2:

The data cube has dimensions region, precipitation, and time.

Part A:

For this question, we would have to do a slice in region we are interested in. After that, we must drill down to month from time dimension and use distributive function sum() over the entries returned by the data cube.

Part B:

Using the same approach with Part A, but this time we will have to use algebraic function avg() over the entries returned by the data cube.

Question 1.3:

The base table is as follows:

RID	Brands	Branch
R1	Audi	Tower Hamlets
R2	Audi	Newham
R3	Audi	Hackney
R4	Ford	Tower Hamlets
R5	Ford	Newham
R6	Ford	Hackney
R7	Mini Cooper	Tower Hamlets
R8	Mini Cooper	Newham
R9	Mini Cooper	Hackney

Bitmap indexing via vehicle brand:

RID	Audi	Ford	Mini Cooper
R1	1	0	0
R2	1	0	0
R3	1	0	0
R4	0	1	0
R5	0	1	0
R6	0	1	0
R7	0	0	1
R8	0	0	1
R9	0	0	1

Bitmap indexing via store branch:

RID	Tower Hamlets	Newham	Hackney
R1	1	0	0
R2	0	1	0
R3	0	0	1
R4	1	0	0
R5	0	1	0
R6	0	0	1
R7	1	0	0
R8	0	1	0
R9	0	0	1

Question 1.4:

The modifications made to the “tutorial_model.json” is shown below. The code is given in full for completeness, any changes made are shown in bold.

```
{
   "dimensions": [
      {
       "name":"item",
       "levels": [
            {
               "name":"category",
               "label":"Category",
               "attributes": ["category", "category_label"]
            },
            {
               "name":"subcategory",
               "label":"Sub-category",
               "attributes": ["subcategory", "subcategory_label"]
            },
            {
               "name":"line_item",
               "label":"Line Item",
               "attributes": ["line_item"]
            }
         ]
      },
      {"name":"year", "role": "time"}
   ],
   "cubes": [
      {
         "name": "ibrd_balance",
         "dimensions": ["item", "year"],
         "measures": [{"name":"amount", "label":"Amount"}],
         "aggregates": [
               {
                  "name": "amount_sum",
                  "function": "sum",
                  "measure": "amount"
               },
               {
                  "name": "amount_min",
                  "function": "min",
                  "measure": "amount"
               },
               {
                  "name": "amount_max",
                  "function": "max",
                  "measure": "amount"
               },
               {
                  "name": "record_count",
                  "function": "count"
               }
            ],
         "mappings": {
                  "item.line_item": "line_item",
                  "item.subcategory": "subcategory",
                  "item.subcategory_label": "subcategory_label",
                  "item.category": "category",
                  "item.category_label": "category_label"
                  },
         "info": {
            "min_date": "2010-01-01",
            "max_date": "2010-12-31"
         }
      }
   ]
}
```

These additional aggregate measures can be used in the following way:

```
browser = workspace.browser(cube)

result = browser.aggregate()

print(result.summary["amount_min"])

print(result.summary["amount_max"])
```

For amount_min and amount_max, a result of -3043 and 128577 are reported, respectively.

Question 1.5:

Part 1:

Part 1 requires a new JSON file to be written. For completeness, the contents of the JSON file is given in full below. **The relevant initialization and setup scripts can be seen in the first code segment displayed on Part 2**. Aggregate functions requested in this question will be displayed in bold:

```
{
    "dimensions": [
       {
        "name":"region",
        "levels": [
              {
                 "name":"region",
                 "label":"Region",
                 "attributes": ["region"]
              }
          ]
       },
       {
        "name":"age",
        "levels": [
              {
                 "name":"age",
                 "label":"Age",
                 "attributes": ["age"]
              }
          ]
```

```
       },
       {
        "name":"online_shopper",
        "levels": [
              {
                 "name":"online_shopper",
                 "label":"Online Shopper",
                 "attributes": ["online_shopper"]
              }
          ]
       }
    ],
    "cubes": [
       {
          "name": "country_income",
          "dimensions": ["region", "age", "online_shopper"],
          "measures": [{"name":"income",
"label":"Income"}],
          "aggregates": [
              {
                 "name": "total_income",
                 "function": "sum",
            "measure": "income"
        },
        {
            "name": "income_min",
            "function": "min",
            "measure": "income"
        },
        {
            "name": "income_max",
            "function": "max",
            "measure": "income"
        },
        {
```

```
                    "name": "income_average",
                    "function": "avg",
                    "measure": "income"
                }
            ],
        "mappings": {
                    "region.region": "region",
                    "age.age": "age",
                    "online_shopper.online_shopper":
"online_shopper"
                    }
        }
    ]
}
```

Part 2:

For this question, the same trend with the tutorial was used along with the JSON file built in Part 1. An engine called **data.sqlite** is generated by the code and the generating code segment is marked with bold. The following code sets up the environment:

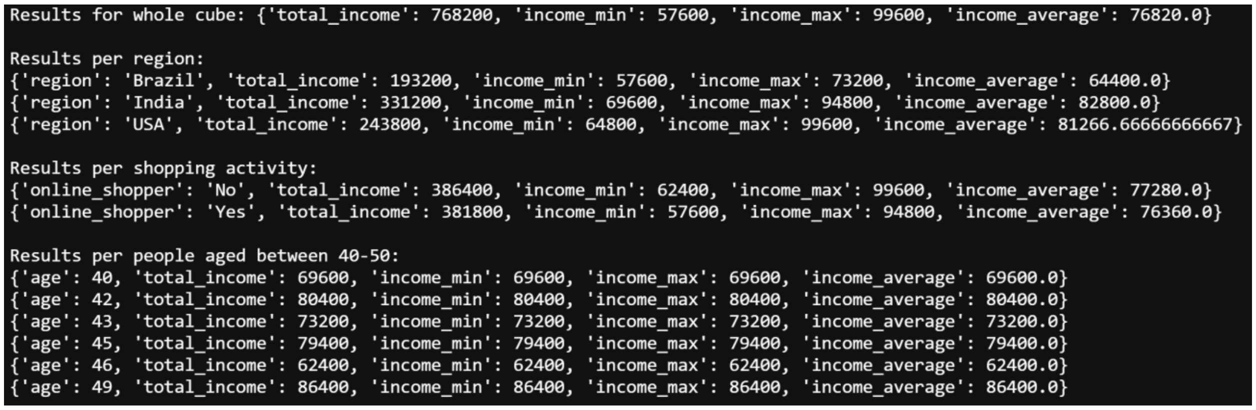
```
from cubes import Workspace

import cubes as cubes

from sqlalchemy import create_engine

from cubes.tutorial.sql import create_table_from_csv

# Follow the same approach from the tutorial: - Part 1

engine = create_engine('sqlite:///data.sqlite')

create_table_from_csv(engine,

                    "country-income.csv",

                    table_name="country_income",

                    fields=[

                        ("region", "string"),

                        ("age", "integer"),

                        ("income", "integer"),

                        ("online_shopper", "string")],

                    create_id=True

                   )

workspace = Workspace()

workspace.register_default_store("sql", url="sqlite:///data.sqlite")

workspace.import_model("country_income.json")

cube = workspace.cube("country_income")

browser = workspace.browser(cube)
```

After the setup, the following code can be used to extract what the question requires:

```
# Aggragate Results for whole cube:

result = browser.aggregate()

print('Results for whole cube: ' + str(result.summary))

print()

# Results per region:

result = browser.aggregate()

result = browser.aggregate(drilldown=["region"])
```

```
print('Results per region:')
for record in result:
    print(record)
print()


# Results per shopping activity:
result = browser.aggregate()
print('Results per shopping activity: ')
result = browser.aggregate(drilldown=["online_shopper"])
for record in result:
    print(record)
print()


# Results per people aged between 40-50:
cuts = [cubes.RangeCut("age", [40], [50])]
cell = cubes.Cell(cube, cuts)
print('Results per people aged between 40-50:')
result = browser.aggregate(cell, drilldown=["age"])
for record in result:
    print(record)
print()
```

The following figure displays the results for the code segment above:



```
Results for whole cube: {'total_income': 768200, 'income_min': 57600, 'income_max': 99600, 'income_average': 76820.0}

Results per region:
{'region': 'Brazil', 'total_income': 193200, 'income_min': 57600, 'income_max': 73200, 'income_average': 64400.0}
{'region': 'India', 'total_income': 331200, 'income_min': 69600, 'income_max': 94800, 'income_average': 82800.0}
{'region': 'USA', 'total_income': 243800, 'income_min': 64800, 'income_max': 99600, 'income_average': 81266.66666666667}

Results per shopping activity:
{'online_shopper': 'No', 'total_income': 386400, 'income_min': 62400, 'income_max': 99600, 'income_average': 77280.0}
{'online_shopper': 'Yes', 'total_income': 381800, 'income_min': 57600, 'income_max': 94800, 'income_average': 76360.0}

Results per people aged between 40-50:
{'age': 40, 'total_income': 69600, 'income_min': 69600, 'income_max': 69600, 'income_average': 69600.0}
{'age': 42, 'total_income': 80400, 'income_min': 80400, 'income_max': 80400, 'income_average': 80400.0}
{'age': 43, 'total_income': 73200, 'income_min': 73200, 'income_max': 73200, 'income_average': 73200.0}
{'age': 45, 'total_income': 79400, 'income_min': 79400, 'income_max': 79400, 'income_average': 79400.0}
{'age': 46, 'total_income': 62400, 'income_min': 62400, 'income_max': 62400, 'income_average': 62400.0}
{'age': 49, 'total_income': 86400, 'income_min': 86400, 'income_max': 86400, 'income_average': 86400.0}
```

Figure 2.Answers to Question 5 – Part B

Part 2: Oct 27, 2020

Question 2.1:

Given each class has 1 item; the item itself becomes the cluster center for that class. To identify the observation's class, one must pick the cluster center which has the smallest distance to the observation. In this case:

For $x_1 = \sqrt{(1-2)^2 + (1-3)^2} = \sqrt{1+4} = \sqrt{5}$

For $x_2 = \sqrt{(-1-2)^2 + (-1-3)^2} = \sqrt{9+16} = 5$

As the observation is closer to x_1, this observation can be assigned to class 0.

Question 2.2:

In k-nearest neighbor classifier, k nearest known datapoints to the observation vote and decide on the observation's class. If the question is a binary classification problem and k is even; there is a chance that k-nearest neighbors have a 50-50 voting for the observations class. In this case a tie breaker is needed. If its odd; the voting will always in favor of a class; hence a separate tie breaking policy is not needed.

Question 2.3:

This follows from our class discussion in Slide 14 in Week 6. If the dataset is imbalanced; meaning the classes are not equally represented in the dataset; the classifier may achieve 99.9 % accuracy but not develop into meaningful models.

In such datasets, the classifier will skew towards the class which has more samples and start to predict everything towards this class. Since the samples for that class is high in number; the accuracy will eventually look high; however, the model may act poorly on the less represented class. It will create an illusion of a well-trained, generalized model in terms of accuracy where it the actual performance is poor implicitly.

Question 2.4:

With perfect precision, the classifier shows no false positives; meaning no negative classes are tagged as positive class. If an observation is tagged as negative by the classifier the prediction is correct; hence can be trusted. (No false positives mean; negative classes will always be predicted correctly).

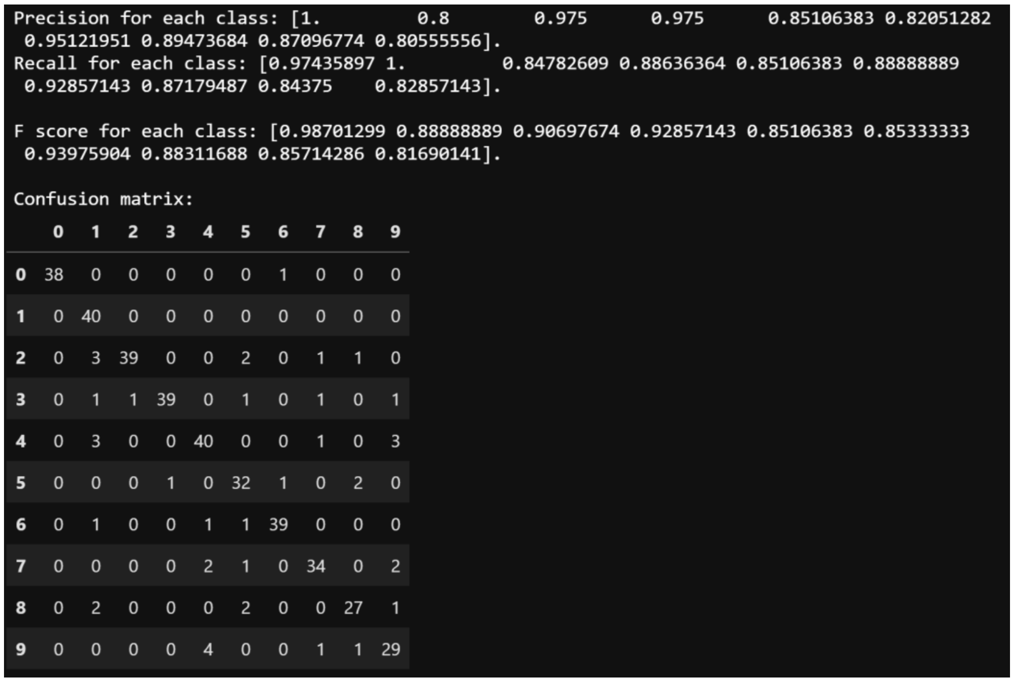
With very low recall, such as 0.1; there are too much false negatives; meaning classifier marks many positives as negatives missing the positive observations. This means it is highly likely that the classifier will tag observations to negative class; even though the actual class is positive; so prediction to positive class cannot be trusted.

In general, such classifier cannot be trusted. It correctly separates out the negative class from the positive class however, mistakes positive classes to negative class. Using F-score as a trust metric we get: $\frac{2PR}{P+R} = 0.18$ which is a low score, and this classifier cannot be **generally** trusted.

Question 2.5:

9th runnable cell in the lab assignment shows the confusion matrix for the 1-nearest neighbor classifier. It is known that the confusion matrix at cell 9 is for 1-nearest neighbor classifier as GridSearchCV operation (at cell 8) indicated that best parameter to use with k-nearest neighbor is 1.

This is the confusion matrix, along with precision, recall and F-score:



```
Precision for each class: [1.         0.8        0.975      0.975      0.85106383 0.82051282
 0.95121951 0.89473684 0.87096774 0.80555556].
Recall for each class: [0.97435897 1.         0.84782609 0.88636364 0.85106383 0.88888889
 0.92857143 0.87179487 0.84375    0.82857143].

F score for each class: [0.98701299 0.88888889 0.90697674 0.92857143 0.85106383 0.85333333
 0.93975904 0.88311688 0.85714286 0.81690141].

Confusion matrix:
```

	0	1	2	3	4	5	6	7	8	9
0	38	0	0	0	0	0	1	0	0	0
1	0	40	0	0	0	0	0	0	0	0
2	0	3	39	0	0	2	0	1	1	0
3	0	1	1	39	0	1	0	1	0	1
4	0	3	0	0	40	0	0	1	0	3
5	0	0	0	1	0	32	1	0	2	0
6	0	1	0	0	1	1	39	0	0	0
7	0	0	0	0	2	1	0	34	0	2
8	0	2	0	0	0	2	0	0	27	1
9	0	0	0	0	4	0	0	1	1	29

Figure 3. Precision, Recall, F-score, and Confusion Matrix for 1-nearest neighbor classifier

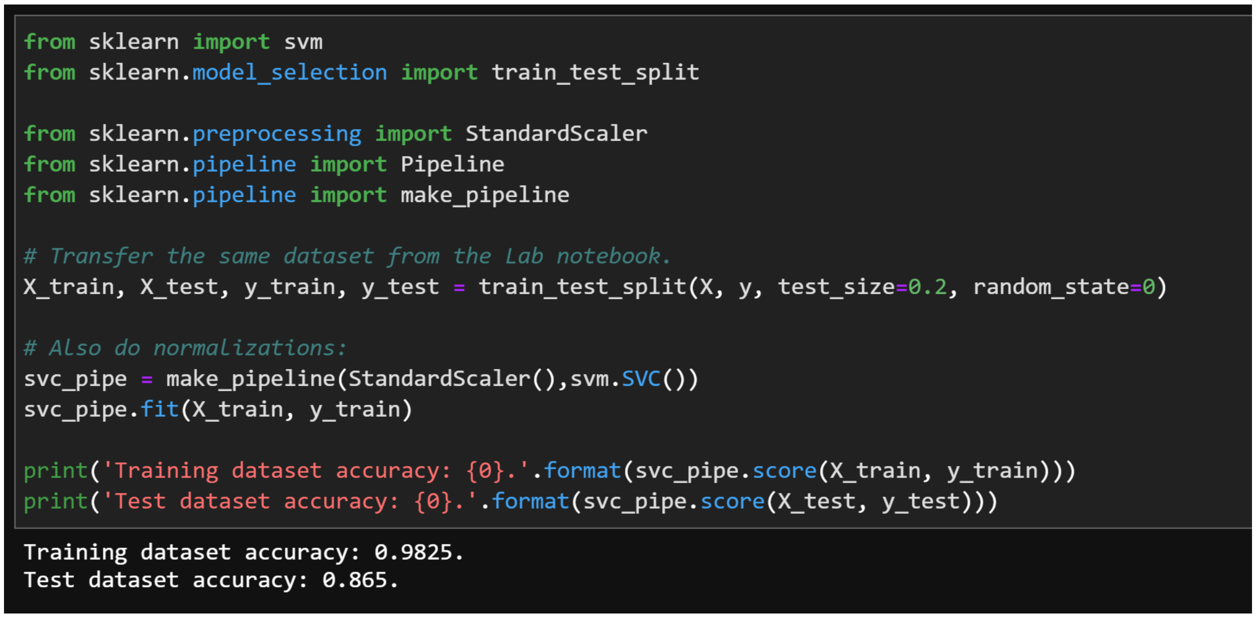
Given the information in Figure 3; it is seen that the most confused classes are 4 and 9 (confused with 4, 7 and 8). As for pairs, most confused pair is 4 with 9.

Question 2.6:

The SVC can be trained with a similar fashion to what is shown in the Lab Assignment tutorial. The code shows the necessary code to construct the environment, it follows from the tutorial:

```
# Using the other notebook to load the dataset: - Some visual parts omitted:

# Configuring the appearance of ``seaborn`` graphics in this notebook

%config InlineBackend.figure_formats = set(['retina'])


import gzip

import pickle

import numpy as np

import pandas as pd
import matplotlib.pyplot as plt

import seaborn as sns

sns.set_style('darkgrid')


# Selecting the training data from the original dataset

f = gzip.open('data/mnist.pkl.gz', 'rb')

X, y = pickle.load(f, encoding='latin1')[0]

f.close()


# Subsampling - Transfer from the Tutorial Part

sample_size = 2000

X, y = X[:sample_size], y[:sample_size]
```

This code does the train and test dataset splits; then uses StandardScaler and SVC from sklearn library to construct a pipeline. Within this pipeline; first normalization is applied to the data, then the data is fed to the SVC. The fit() method can be used to do the training. The following is the code for this section along with training and testing accuracy achieved from the code:



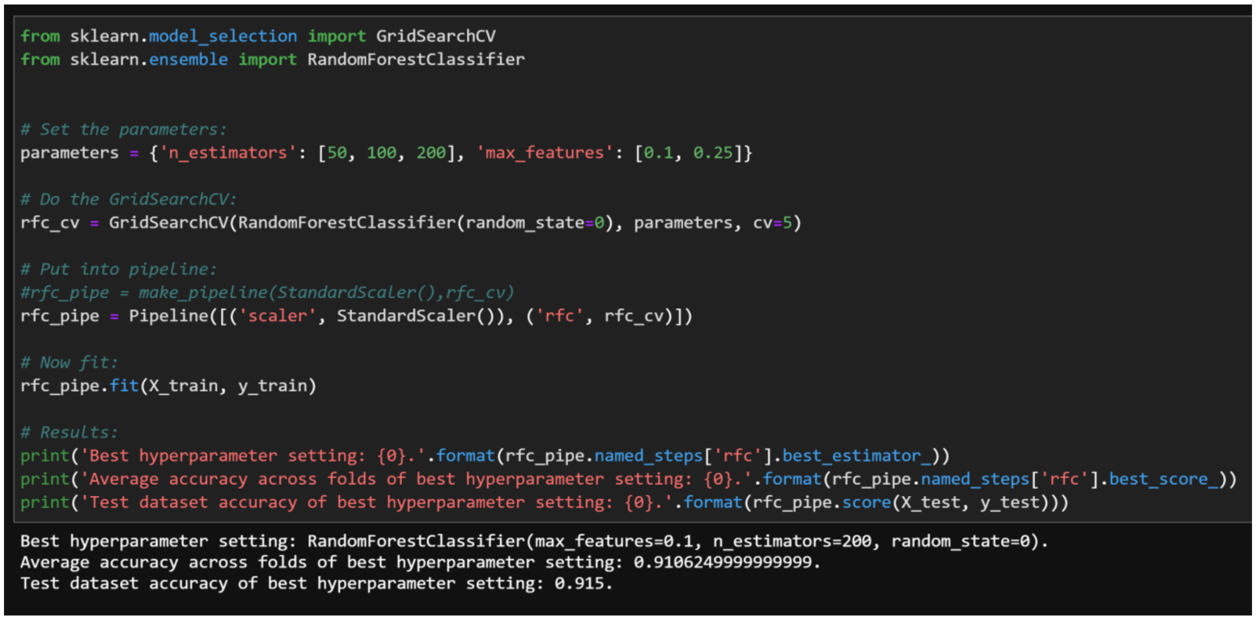
```
from sklearn import svm
from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline

# Transfer the same dataset from the Lab notebook.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Also do normalizations:
svc_pipe = make_pipeline(StandardScaler(),svm.SVC())
svc_pipe.fit(X_train, y_train)

print('Training dataset accuracy: {0}.'.format(svc_pipe.score(X_train, y_train)))
print('Test dataset accuracy: {0}.'.format(svc_pipe.score(X_test, y_test)))
```

```
Training dataset accuracy: 0.9825.
Test dataset accuracy: 0.865.
```

Figure 4. Code and Results for Question 2.6

Question 2.7:

The first code segment in Question 2.6 is also used for this section to initialize the environment. The following code is written for this question. The asked results is also displayed below:



```python
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier


# Set the parameters:
parameters = {'n_estimators': [50, 100, 200], 'max_features': [0.1, 0.25]}

# Do the GridSearchCV:
rfc_cv = GridSearchCV(RandomForestClassifier(random_state=0), parameters, cv=5)

# Put into pipeline:
#rfc_pipe = make_pipeline(StandardScaler(),rfc_cv)
rfc_pipe = Pipeline([('scaler', StandardScaler()), ('rfc', rfc_cv)])

# Now fit:
rfc_pipe.fit(X_train, y_train)

# Results:
print('Best hyperparameter setting: {0}.'.format(rfc_pipe.named_steps['rfc'].best_estimator_))
print('Average accuracy across folds of best hyperparameter setting: {0}.'.format(rfc_pipe.named_steps['rfc'].best_score_))
print('Test dataset accuracy of best hyperparameter setting: {0}.'.format(rfc_pipe.score(X_test, y_test)))
```

```
Best hyperparameter setting: RandomForestClassifier(max_features=0.1, n_estimators=200, random_state=0).
Average accuracy across folds of best hyperparameter setting: 0.9106249999999999.
Test dataset accuracy of best hyperparameter setting: 0.915.
```

Figure 5. Answer to Question 2.7

Parameters are written in a dictionary form following from the tutorial. First the grid search cross validation is initialized then put into a pipeline using StandardScaler() to normalize data before proceeding with training.

Question 2.8:

For reference the function is given below, note that the lines are labeled:

```
def kmeans_update(X, cluster_centers): # Line 1
    y_pred = np.argmin(cdist(X, cluster_centers), axis=1) # Line 2


    next_cluster_centers = np.zeros(cluster_centers.shape) # Line 3
    for i in range(len(next_cluster_centers)): # Line 4
        next_cluster_centers[i] = X[y_pred == i].mean(axis=0) # Line 5


    return y_pred, next_cluster_centers # Line 6
```

Line 1 is the function definition. It gets unlabeled data matrix and current cluster center information respectively. The cluster centers are initially defined randomly. Number of clusters are implicitly expressed within dimensionality of the cluster_centers.

Line 2 uses cdist to compute the spatical distance between all datapoints of X along axis 1. Spatial distance is Euclidean by default. It also assigns each sample to a class according to its closest cluster center using argmin operation.

Line 3 creates an empty array of cluster_centers, data labeled in Line 2 will be used to move the cluster centers.

Line 4 does this operation : “For every cluster center do the following:”

Line 5 does conditional indexing to fetch all samples belonging to a current class, calculate their mean. This mean is assigned as the new cluster center.

Line 6 returns the labels generated with this update and the new set of cluster centers achieved by this function.

Basically, these 6 lines runs the Expectation Maximization algorithm once. For k-means; this algorithm is called many times until cluster centers converge or computational budget is reached.