Bonus 11

For this Bonus assignment I tried to use our Kmeans implementation to predict the titanic dataset.

First I replaced the scaling with the standardscaler of sklearn. I then slightly modified the prediction function to use panda dataframes so it would be easier to work with. The predict function has been further altered to create a Kaggle submission set each time its run.

The accuracy on the training data is arrount 80% but the score on Kaggle is considerably lower compared to for example the extratree or random forest or sgboost.

I experimented with different nodes and runtimes but the limitation of 10 submissions per day is a limiting factor.

The accuracy achieved with 100 nodes and 10 iterations was 62%.

I also tried to follow a tutorial that implemented a kmeans method but on a previously prepared dataset. Using this method a score of 79 on Kaggle was achieved with the K means algorithm. See Appendix 2

Appendix 1

import numpy as np

import sklearn as sk

from sklearn.cluster import KMeans

from sklearn.metrics import accuracy\_score, confusion\_matrix

from sklearn.mixture import GaussianMixture

import matplotlib.pyplot as plt

from typing import Union

import os

import numpy as np

from numpy.core.function\_base import linspace

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import (train\_test\_split,

                                     RandomizedSearchCV)

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

from sklearn.metrics import (confusion\_matrix, accuracy\_score, recall\_score,

                             precision\_score)

from sklearn.preprocessing import StandardScaler

from scipy.stats import uniform as sp\_randFloat

from scipy.stats import randint as sp\_randInt

from sklearn import svm

from sklearn.neighbors import KNeighborsClassifier

from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier

from sklearn.ensemble import VotingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from itertools import product

from sklearn.ensemble import VotingClassifier

from sklearn.ensemble import AdaBoostClassifier

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import GridSearchCV

from sklearn.feature\_selection import SelectFromModel

from sklearn.linear\_model import LogisticRegression, LogisticRegressionCV

from sklearn.model\_selection import StratifiedKFold

from tools import load\_iris, image\_to\_numpy, plot\_gmm\_results

def get\_even\_better\_titanic():

    '''

    Loads the cleaned titanic dataset but change

    how we handle the age column.

    Loads the cleaned titanic dataset

    '''

    # Load in the raw data

    # check if data directory exists for Mimir submissions

    # DO NOT REMOVE

    if os.path.exists('./10\_boosting/data/train.csv'):

        train = pd.read\_csv('./10\_boosting/data/train.csv')

        test = pd.read\_csv('./10\_boosting/data/test.csv')

    else:

        train = pd.read\_csv('train.csv')

        test = pd.read\_csv('test.csv')

    # Concatenate the train and test set into a single dataframe

    # we drop the `Survived` column from the train set

    X\_full = pd.concat([train.drop('Survived', axis=1), test], axis=0)

    # The cabin category consist of a letter and a number.

    # We can divide the cabin category by extracting the first

    # letter and use that to create a new category. So before we

    # drop the `Cabin` column we extract these values

    X\_full['Cabin\_mapped'] = X\_full['Cabin'].astype(str).str[0]

    # Then we transform the letters into numbers

    cabin\_dict = {k: i for i, k in enumerate(X\_full.Cabin\_mapped.unique())}

    X\_full.loc[:, 'Cabin\_mapped'] =\

        X\_full.loc[:, 'Cabin\_mapped'].map(cabin\_dict)

    Title\_Dictionary = {

        "Capt": "Officer",

        "Col": "Officer",

        "Major": "Officer",

        "Jonkheer": "Royalty",

        "Don": "Royalty",

        "Sir" : "Royalty",

        "Dr": "Officer",

        "Rev": "Officer",

        "the Countess":"Royalty",

        "Mme": "Mrs",

        "Mlle": "Miss",

        "Ms": "Mrs",

        "Mr" : "Mr",

        "Mrs" : "Mrs",

        "Miss" : "Miss",

        "Master" : "Master",

        "Lady" : "Royalty"}

    X\_full['Title'] = X\_full['Name'].map(lambda name:name.split(',')[1].split('.')[0].strip())

    # a map of more aggregated title

    # we map each title

    X\_full['Title'] = X\_full.Title.map(Title\_Dictionary)

    # age

    grouped\_train = X\_full.iloc[:891].groupby(['Sex','Pclass','Title'])

    grouped\_median\_train = grouped\_train.median()

    grouped\_median\_train = grouped\_median\_train.reset\_index()[['Sex', 'Pclass', 'Title', 'Age']]

    grouped\_median\_train.head()

    def fill\_age(row):

        condition = (

            (grouped\_median\_train['Sex'] == row['Sex']) &

            (grouped\_median\_train['Title'] == row['Title']) &

            (grouped\_median\_train['Pclass'] == row['Pclass'])

        )

        return grouped\_median\_train[condition]['Age'].values[0]

        # a function that fills the missing values of the Age variable

    X\_full['Age'] = X\_full.apply(lambda row: fill\_age(row) if np.isnan(row['Age']) else row['Age'], axis=1)

    #clean the Name variable

    X\_full.drop('Name', axis=1, inplace=True)

    # encoding in dummy variable

    titles\_dummies = pd.get\_dummies(X\_full['Title'], prefix='Title')

    X\_full = pd.concat([X\_full, titles\_dummies], axis=1)

    # removing the title variable

    X\_full.drop('Title', axis=1, inplace=True)

    # add missing fares

    X\_full.Fare.fillna(X\_full.iloc[:891].Fare.mean(), inplace=True)

    # We drop multiple columns that contain a lot of NaN values

    # in this assignment

    # Maybe we should

    X\_full.drop(

        ['PassengerId'],

        inplace=True, axis=1)

    # Instead of dropping the Embarked column we replace NaN values

    # with `S` denoting Southampton, the most common embarking

    # location

    X\_full['Embarked'].fillna('S', inplace=True)

    embarked\_dummies = pd.get\_dummies(X\_full['Embarked'], prefix='Embarked')

    X\_full = pd.concat([X\_full, embarked\_dummies], axis=1)

    X\_full.drop('Embarked', axis=1, inplace=True)

    X\_full.Cabin.fillna('U', inplace=True)

    # mapping each Cabin value with the cabin letter

    X\_full['Cabin'] = X\_full['Cabin'].map(lambda c: c[0])

    # dummy encoding ...

    cabin\_dummies = pd.get\_dummies(X\_full['Cabin'], prefix='Cabin')

    X\_full = pd.concat([X\_full, cabin\_dummies], axis=1)

    X\_full.drop('Cabin', axis=1, inplace=True)

    X\_full['Sex'] = X\_full['Sex'].map({'male':1, 'female':0})

    # We then use the get\_dummies function to transform text

    # and non-numerical values into binary categories.

    X\_dummies = pd.get\_dummies(

        X\_full,

        columns=['Cabin\_mapped'],

        drop\_first=True)

    # encoding into 3 categories:

    pclass\_dummies = pd.get\_dummies(X\_full['Pclass'], prefix="Pclass")

    # adding dummy variable

    X\_full = pd.concat([X\_full, pclass\_dummies],axis=1)

    # removing "Pclass"

    X\_full.drop('Pclass',axis=1,inplace=True)

    # a function that extracts each prefix of the ticket, returns 'XXX' if no prefix (i.e the ticket is a digit)

    def cleanTicket(ticket):

        ticket = ticket.replace('.','')

        ticket = ticket.replace('/','')

        ticket = ticket.split()

        ticket = map(lambda t : t.strip(), ticket)

        ticket = filter(lambda t : not t.isdigit(), ticket)

        ticketlist = list(filter(lambda t : not t.isdigit(), ticket))

        if len(ticketlist) > 0:

            return ticketlist[0]

        else:

            return 'XXX'

    # Extracting dummy variables from tickets:

    X\_full['Ticket'] = X\_full['Ticket'].map(cleanTicket)

    tickets\_dummies = pd.get\_dummies(X\_full['Ticket'], prefix='Ticket')

    X\_full = pd.concat([X\_full, tickets\_dummies], axis=1)

    X\_full.drop('Ticket', inplace=True, axis=1)

    X\_full['FamilySize'] = X\_full['Parch'] + X\_full['SibSp'] + 1

    # introducing other features based on the family size

    X\_full['Singleton'] = X\_full['FamilySize'].map(lambda s: 1 if s == 1 else 0)

    X\_full['SmallFamily'] = X\_full['FamilySize'].map(lambda s: 1 if 2 <= s <= 4 else 0)

    X\_full['LargeFamily'] = X\_full['FamilySize'].map(lambda s: 1 if 5 <= s else 0)

    # We now have the cleaned data we can use in the assignment

    X = X\_full[:len(train)]

    submission\_X = X\_full[len(train):]

    y = train.Survived

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(

        X, y, test\_size=.3, random\_state=5, stratify=y)

    return (X\_train, y\_train), (X\_test, y\_test), submission\_X

def get\_better\_titanic():

    '''

    Loads the cleaned titanic dataset but change

    how we handle the age column.

    Loads the cleaned titanic dataset

    '''

    # Load in the raw data

    # check if data directory exists for Mimir submissions

    # DO NOT REMOVE

    if os.path.exists('./10\_boosting/data/train.csv'):

        train = pd.read\_csv('./10\_boosting/data/train.csv')

        test = pd.read\_csv('./10\_boosting/data/test.csv')

    else:

        train = pd.read\_csv('train.csv')

        test = pd.read\_csv('test.csv')

    # Concatenate the train and test set into a single dataframe

    # we drop the `Survived` column from the train set

    X\_full = pd.concat([train.drop('Survived', axis=1), test], axis=0)

    # The cabin category consist of a letter and a number.

    # We can divide the cabin category by extracting the first

    # letter and use that to create a new category. So before we

    # drop the `Cabin` column we extract these values

    X\_full['Cabin\_mapped'] = X\_full['Cabin'].astype(str).str[0]

    # Then we transform the letters into numbers

    cabin\_dict = {k: i for i, k in enumerate(X\_full.Cabin\_mapped.unique())}

    X\_full.loc[:, 'Cabin\_mapped'] =\

        X\_full.loc[:, 'Cabin\_mapped'].map(cabin\_dict)

    # We drop multiple columns that contain a lot of NaN values

    # in this assignment

    # Maybe we should

    X\_full.drop(

        ['PassengerId', 'Cabin', 'Name', 'Ticket'],

        inplace=True, axis=1)

    # Instead of dropping the fare column we replace NaN values

    # with the 3rd class passenger fare mean.

    fare\_mean = X\_full[X\_full.Pclass == 3].Fare.mean()

    X\_full['Fare'].fillna(fare\_mean, inplace=True)

    # Instead of dropping the Embarked column we replace NaN values

    # with `S` denoting Southampton, the most common embarking

    # location

    X\_full['Embarked'].fillna('S', inplace=True)

    # We then use the get\_dummies function to transform text

    # and non-numerical values into binary categories.

    X\_dummies = pd.get\_dummies(

        X\_full,

        columns=['Sex', 'Cabin\_mapped', 'Embarked'],

        drop\_first=True)

    X\_dummies['Age'].fillna(X\_dummies.Age.mean(), inplace=True)

    '''

    X\_Age = X\_dummies[X\_dummies.Age.notna()].drop(['Age'], axis=1)

    y\_Age = X\_dummies[X\_dummies.Age.notna()]['Age']

    X\_pred = X\_dummies[X\_dummies.Age.isna()].drop(['Age'], axis=1)

    print(X\_dummies[X\_dummies.Age.notna()].drop(['Age'], axis=1))

    from sklearn import svm

    c = svm.SVR(kernel='rbf')

    c.fit(X\_Age,y\_Age)

    #X\_dummies[X\_dummies['Age'].isna()]['Age'] = c.predict(X\_pred)

    X\_dummies.loc[X\_dummies.Age.isna(),'Age'] = c.predict(X\_pred)

    '''

    # We now have the cleaned data we can use in the assignment

    X = X\_dummies[:len(train)]

    submission\_X = X\_dummies[len(train):]

    y = train.Survived

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(

        X, y, test\_size=.3, random\_state=5, stratify=y)

    #print(y)

    return (X\_train, y\_train), (X\_test, y\_test), submission\_X

def get\_titanic():

    '''

    Loads the cleaned titanic dataset

    '''

    # Load in the raw data

    # check if data directory exists for Mimir submissions

    # DO NOT REMOVE

    if os.path.exists('./10\_boosting/data/train.csv'):

        train = pd.read\_csv('./10\_boosting/data/train.csv')

        test = pd.read\_csv('./10\_boosting/data/test.csv')

    else:

        train = pd.read\_csv('train.csv')

        test = pd.read\_csv('test.csv')

    # Concatenate the train and test set into a single dataframe

    # we drop the `Survived` column from the train set

    X\_full = pd.concat([train.drop('Survived', axis=1), test], axis=0)

    # The cabin category consist of a letter and a number.

    # We can divide the cabin category by extracting the first

    # letter and use that to create a new category. So before we

    # drop the `Cabin` column we extract these values

    X\_full['Cabin\_mapped'] = X\_full['Cabin'].astype(str).str[0]

    # Then we transform the letters into numbers

    cabin\_dict = {k: i for i, k in enumerate(X\_full.Cabin\_mapped.unique())}

    X\_full.loc[:, 'Cabin\_mapped'] =\

        X\_full.loc[:, 'Cabin\_mapped'].map(cabin\_dict)

    # We drop multiple columns that contain a lot of NaN values

    # in this assignment

    # Maybe we should

    X\_full.drop(

        ['PassengerId', 'Cabin', 'Age', 'Name', 'Ticket'],

        inplace=True, axis=1)

    # Instead of dropping the fare column we replace NaN values

    # with the 3rd class passenger fare mean.

    fare\_mean = X\_full[X\_full.Pclass == 3].Fare.mean()

    X\_full['Fare'].fillna(fare\_mean, inplace=True)

    # Instead of dropping the Embarked column we replace NaN values

    # with `S` denoting Southampton, the most common embarking

    # location

    X\_full['Embarked'].fillna('S', inplace=True)

    # We then use the get\_dummies function to transform text

    # and non-numerical values into binary categories.

    X\_dummies = pd.get\_dummies(

        X\_full,

        columns=['Sex', 'Cabin\_mapped', 'Embarked'],

        drop\_first=True)

    # We now have the cleaned data we can use in the assignment

    X = X\_dummies[:len(train)]

    submission\_X = X\_dummies[len(train):]

    y = train.Survived

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(

        X, y, test\_size=.3, random\_state=5, stratify=y)

    return (X\_train, y\_train), (X\_test, y\_test), submission\_X

def build\_kaggle\_submission(prediction):

    '''

    Given a prediction, this function will build a

    kaggle compatible submission and save it to disk

    at ./data/your\_submission.csv`.

    '''

    test = pd.read\_csv('./10\_boosting/data/test.csv')

    submission = pd.concat(

        [test.PassengerId, pd.DataFrame(prediction)],

        axis='columns')

    submission.columns = ["PassengerId", "Survived"]

    submission.to\_csv('./11\_k\_means/your\_submission.csv', header=True, index=False)

    print('Your submission can be found at ./11\_k\_means/your\_submission.csv')

def distance\_matrix(

    X: np.ndarray,

    Mu: np.ndarray

) -> np.ndarray:

    '''

    Returns a matrix of euclidian distances between points in

    X and Mu.

    Input arguments:

    \* X (np.ndarray): A [n x f] array of samples

    \* Mu (np.ndarray): A [k x f] array of prototypes

    Returns:

    out (np.ndarray): A [n x k] array of euclidian distances

    where out[i, j] is the euclidian distance between X[i, :]

    and Mu[j, :]

    '''

    n,f = X.shape

    k,f = Mu.shape

    distances = np.zeros((n,k))

    for i in range(n):

        for j in range(k):

            distances[i,j] = np.sqrt(sum(np.power(X[i,:] - Mu[j,:],2)))

    return distances

def determine\_r(dist: np.ndarray) -> np.ndarray:

    '''

    Returns a matrix of binary indicators, determining

    assignment of samples to prototypes.

    Input arguments:

    \* dist (np.ndarray): A [n x k] array of distances

    Returns:

    out (np.ndarray): A [n x k] array where out[i, j] is

    1 if sample i is closest to prototype j and 0 otherwise.

    '''

    R = np.zeros((dist.shape))

    for i in range(dist.shape[0]):

        R[i,np.argmin(dist[i,:])] = 1

    return R

def determine\_j(R: np.ndarray, dist: np.ndarray) -> float:

    '''

    Calculates the value of the objective function given

    arrays of indicators and distances.

    Input arguments:

    \* R (np.ndarray): A [n x k] array where out[i, j] is

        1 if sample i is closest to prototype j and 0

        otherwise.

    \* dist (np.ndarray): A [n x k] array of distances

    Returns:

    \* out (float): The value of the objective function

    '''

    J = np.zeros((dist.shape))

    for n in range(dist.shape[0]):

        for k in range(dist.shape[1]):

            J[n, k] = R[n, k] \* dist[n, k]

    res = sum(sum(J))/J.shape[0]

    return res

def update\_Mu(

    Mu: np.ndarray,

    X: np.ndarray,

    R: np.ndarray

) -> np.ndarray:

    '''

    Updates the prototypes, given arrays of current

    prototypes, samples and indicators.

    Input arguments:

    Mu (np.ndarray): A [k x f] array of current prototypes.

    X (np.ndarray): A [n x f] array of samples.

    R (np.ndarray): A [n x k] array of indicators.

    Returns:

    out (np.ndarray): A [k x f] array of updated prototypes.

    '''

    N, K = R.shape

    newmu = np.zeros\_like(Mu)

    for k in range(K):

        res1=0

        res2=0

        for n in range(N):

            res1 += R[n,k]\*X[n]

            res2 += R[n,k]

        if res2 != 0:

            newmu[k] = res1/res2

        else:

            newmu[k] = Mu[k]

    return newmu

    ...

def k\_means(

    X\_standard: np.ndarray,

    k: int,

    num\_its: int

) -> Union[list, np.ndarray, np.ndarray]:

    # We first have to standardize the samples

    # run the k\_means algorithm on X\_st, not X.

    # we pick K random samples from X as prototypes

    nn = sk.utils.shuffle(range(X\_standard.shape[0]))

    Mu = X\_standard[nn[0: k], :]

    Js = []

    for i in range(num\_its):

        distances = distance\_matrix(X\_standard, Mu)

        R = determine\_r(distances)

        J = determine\_j(R, distances)

        Js.append(J)

        Mu = update\_Mu(Mu, X\_standard, R)

    # Then we have to "de-standardize" the prototypes

    '''

    for i in range(k):

        Mu[i, :] = Mu[i, :] \* np.sqrt(ss.var\_) + ss.mean\_

    '''

    return Mu, R, Js

def \_plot\_j():

    Mu, R, Js = k\_means(X, 4, 10)

    plt.clf

    plt.plot(Js)

    plt.title("Evolution of J")

    plt.ylabel("Value of J")

    plt.xlabel("Iteration")

    plt.savefig("11\_k\_means/1\_6\_1.png")

    plt.show()

def \_plot\_multi\_j():

    K= [2,3,5,10]

    plt.clf

    for k in K:

        Mu, R, Js = k\_means(tr\_X.to\_numpy(), k, 50)

        print(Js)

        plt.plot(Js, label='k = {}'.format(k))

    plt.title("Evolution of J")

    plt.ylabel("Value of J")

    plt.xlabel("Iteration")

    plt.legend(loc='upper center')

    plt.show()

def k\_means\_predict(

    X: np.ndarray,

    t: np.ndarray,

    classes: list,

    num\_its: int,

    k:int

) -> np.ndarray:

    '''

    Determine the accuracy and confusion matrix

    of predictions made by k\_means on a dataset

    [X, t] where we assume the most common cluster

    for a class label corresponds to that label.

    Input arguments:

    \* X (np.ndarray): A [n x f] array of input features

    \* t (np.ndarray): A [n] array of target labels

    \* classes (list): A list of possible target labels

    \* num\_its (int): Number of k\_means iterations

    Returns:

    \* the predictions (list)

    '''

    ss = StandardScaler()

    Xstd =ss.fit\_transform(X)

    Xtststd = ss.transform(tst\_X)

    Xsubstd = ss.transform(submission\_X)

    nc = len(classes)

    nt = len(t)

    Mu, R, Js = k\_means(Xstd, k=k, num\_its=num\_its)

    R = pd.DataFrame(np.argmax(R, axis=1), columns = ['Cluster'])

    R['Survived'] = t

    survival\_mapper = R.groupby('Cluster')['Survived'].mean().round().astype(np.int).to\_dict()

    R['group\_survived'] = R.Cluster.map(survival\_mapper)

    survival\_map = R.groupby('Cluster')['Survived'].mean().round().astype(np.int).to\_dict()

    print((R['Survived'] == R['group\_survived']).mean())

    distances = distance\_matrix(Xsubstd, Mu)

    R\_sub = determine\_r(distances)

    predict = pd.DataFrame(np.argmax(R\_sub, axis=1), columns = ['Cluster'])

    test = pd.read\_csv('./10\_boosting/data/test.csv')

    predict['Survived'] = predict.Cluster.map(survival\_map)

    predict['PassengerId'] = test['PassengerId']

    predict = predict[['PassengerId', 'Survived']].sort\_values('PassengerId').reset\_index(drop=True)

    return R, predict

def \_iris\_kmeans\_accuracy():

    y\_pred = k\_means\_predict(X, y, c, 5)

    print(accuracy\_score(y,y\_pred))

    print(confusion\_matrix(y,y\_pred))

def \_my\_kmeans\_on\_image():

    ...

def plot\_image\_clusters(n\_clusters: int):

    '''

    Plot the clusters found using sklearn k-means.

    '''

    image, (w, h) = image\_to\_numpy()

    c = KMeans(n\_clusters=n\_clusters)

    c.fit(image)

    print(w, h)

    plt.subplot('121')

    plt.title("Original image")

    plt.imshow(image.reshape(w, h, 3))

    plt.ylabel("Pixels")

    plt.xlabel("Pixels")

    plt.subplot('122')

    plt.title("Plot using {} clusters".format(n\_clusters))

    # uncomment the following line to run

    plt.xlabel("Pixels")

    plt.imshow(c.labels\_.reshape(w, h), cmap="plasma")

    plt.savefig("11\_k\_means/2\_1\_{}.png".format(n\_clusters))

    plt.show()

def \_gmm\_info():

    c = GaussianMixture(n\_components=3)

    c.fit(X)

    print(c.means\_)

    print(c.covariances\_)

    print(c.weights\_)

def \_plot\_gmm():

    c = GaussianMixture(n\_components=3)

    c.fit(X)

    plot\_gmm\_results(X,c.predict(X),c.means\_,c.covariances\_)

def kmeans\_sk(

    X: np.ndarray,

    t: np.ndarray,

    classes: list,

    num\_its: int,

    k:int

) -> np.ndarray:

    ss = StandardScaler()

    X\_std = ss.fit\_transform(tr\_X)

    Xsubs = ss.fit\_transform(submission\_X)

    X\_tststd = ss.fit\_transform(tst\_X)

    c = KMeans(n\_clusters=k, max\_iter=num\_its)

    c.fit(X\_std, tst\_y)

    clst = c.predict(X\_tststd)

    R = pd.DataFrame(clst, columns = ['Cluster'])

    R['Survived'] = tr\_y

    survival\_mapper = R.groupby('Cluster')['Survived'].mean().round().astype(np.int).to\_dict()

    R['group\_survived'] = R.Cluster.map(survival\_mapper)

    accuracy = (R['Survived'] == R['group\_survived']).mean()

    print(accuracy)

(tr\_X, tr\_y), (tst\_X, tst\_y), submission\_X = get\_titanic()

#\_plot\_multi\_j()

pred, predict = k\_means\_predict(tr\_X.to\_numpy(), tr\_y.to\_numpy(), [0,1], 20, 70)

build\_kaggle\_submission(predict['Survived'])

'''

accuracy = (pred['Survived'] == pred['group\_survived']).mean()

print(accuracy)

pred = kmeans\_sk(tr\_X, tr\_y, [0,1], 150, 47)

'''

Appendix 2

# To add a new cell, type '# %%'

# To add a new markdown cell, type '# %% [markdown]'

# %% [markdown]

# # k-means-clustering from scratch

# In This notebook we write our own k-means from scratch in Python and apply it to predict the survivability on the famous Titanic dataset. The method is very similar to k-nearest-neighbor, however now we find groups of similar data.

#

# ### Content

# 1. Create a k-means algorithm

# 2. Predict Titanic Survivabillity

# 3. Solution using Scikit-Learn

# %%

# %%

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# %% [markdown]

# # 1. Create a k-means algorithm

# %% [markdown]

# Lets first generate a toy dataset to apply k-means on:

# %%

lower, higher = -10, 10

points = 100

clusters = 5

width = 1.2

random\_state = 1

np.random.seed(random\_state)

data = []

for ix in range(clusters):

    cx, cy = np.random.randint(lower, higher), np.random.randint(lower, higher),

    x = np.random.normal(cx, width, size=(points,))

    y = np.random.normal(cy, width, size=(points,))

    data.append(

        pd.DataFrame({

            'x': x,

            'y': y,

            'label': ix + 1,

        })

    )

data = pd.concat(data)

X = data.drop('label', axis=1)

y = data.drop(['x', 'y'], axis=1)

fig, ax = plt.subplots(1,1, figsize=(10,6))

ax.set\_aspect('equal')

plt.axis('off')

fig.tight\_layout()

for group, group\_data in data.groupby('label'):

    plt.plot(group\_data.x, group\_data.y, 'o', ms=5, label=f'Class {group}')

lgd = ax.legend(loc=8, fontsize=20, frameon=False, markerscale=2)

# %% [markdown]

# Next, we need to define a distance function. A very common function is the Euclidean distance.

# %%

def euclidean\_distance(vector1, vector2):

    return np.sqrt(np.sum((vector1 - vector2)\*\*2))

# test function

vec1 = np.array([3, 0])

vec2 = np.array([0, 4])

# this is the 3:4:5 triangle and therefore, it should return 5 (Long live Pythagoras)

euclidean\_distance(vec1, vec2)

# %% [markdown]

# The euclidean distance calculates the square root, however, the squared distance is very similar and computationally a bit easier.

# %%

def squared\_difference(dataset, vector, columns=['x', 'y']):

    return ((dataset[columns[0]] - vector[columns[0]])\*\*2 +

                  (dataset[columns[1]] - vector[columns[1]])\*\*2)

# %% [markdown]

# As we already know the amount of clusters, lets just set that. For k-means we need a starting point. We will randomly select the number of clusters amount of points from our dataset.

# %%

number\_of\_clusters = len(y.label.unique())

centroids = X.sample(number\_of\_clusters, random\_state=random\_state)

number\_of\_clusters

# %% [markdown]

# Next we need a function to cluster our dataset according to the center points we have randomly chosen.

# %%

def cluster\_dataset(dataset, centroids):

    distances = pd.concat([

        ((dataset - centroid)\*\*2).sum(axis=1)

        for ix, centroid in centroids.iterrows()],

        axis=1,

    )

    return dataset.assign( cluster = distances.idxmin(axis=1)), distances.min(axis=1).sum()

# %%

clustered, \_ = cluster\_dataset(X, centroids)

# %%

clustered.cluster.unique()

# %% [markdown]

# Lets plot our clustered dataset and include the centers:

# %%

fig, ax = plt.subplots(1,1, figsize=(10,6))

ax.set\_aspect('equal')

plt.axis('off')

fig.tight\_layout()

for group, group\_data in clustered.groupby('cluster'):

    plt.plot(group\_data.x, group\_data.y, 'o', ms=5, label=f'Class {group + 1}')

for ix, centroid in centroids.iterrows():

    ax.plot(centroid.x, centroid.y, 'ro', ms=10)

lgd = ax.legend(loc=8, fontsize=20, frameon=False, markerscale=2)

# xlim = ax.set\_xlim([-5, 26])

# %% [markdown]

# The next step in k-means is to update the centroids to have the center averaged over the found cluster. This will move the centroids to the new position and we can repeat this step to converge to the perfect position.

# %%

def update\_centroids(clustered\_dataset):

    new\_centroids = clustered\_dataset.groupby('cluster').mean().reset\_index(drop=True)

    return new\_centroids

# %%

centroids = update\_centroids(clustered)

# %%

centroids

# %% [markdown]

# We can visualize this in steps:

# %%

from time import sleep

from IPython.display import clear\_output

def plot(clustered, centroids, iteration, total\_sum):

    clear\_output(wait=True)

    fig, ax = plt.subplots(1,1, figsize=(10,6))

    ax.set\_aspect('equal')

    plt.axis('off')

    fig.tight\_layout()

    for group, group\_data in clustered.groupby('cluster'):

        ax.plot(group\_data.x, group\_data.y, 'o', ms=5, label=f'Class {group + 1}')

    for ix, centroid in centroids.iterrows():

        ax.plot(centroid.x, centroid.y, 'ro', ms=15)

    lgd = ax.legend(loc=8, fontsize=20, frameon=False, markerscale=2)

    ax.set\_title(f'Iteration {iteration} (sum = {total\_sum})')

    plt.show()

def cluster\_data(dataset, number\_of\_clusters, max\_iter=20, show=False, pause=0.5):

    ds = dataset.copy()

    centroids = ds.sample(number\_of\_clusters, random\_state=random\_state)

    previous\_sum = pd.Series(range(len(centroids)))

    for iteration in range(max\_iter):

        clustered, total\_sum = cluster\_dataset(ds, centroids)

        centroids = update\_centroids(clustered)

        if show:

            plot(clustered, centroids, iteration + 1, total\_sum)

            sleep(pause)

        if (total\_sum - previous\_sum).sum() == 0:  # this is not the best method

            break

        previous\_sum = total\_sum

    return clustered, centroids, total\_sum

clust, cent, total\_sum = cluster\_data(X, number\_of\_clusters= 5, show=True)

# %% [markdown]

# This is a great method to identify groups, however we need to provide the amount of clusters. One way to select this amount is using the Elbow method. This is a visual method in which we plot the average distance of all datapoints to their centroids as function of number of clusters (k). These will go down fast in the beginning and eventually converge to a flat line. This is because we get clusters with only one or a few points and it cannot get much better. The "elbow" of the graph, i.e. the corner is approximately the best k-value. For this, we do need to have the proper distance, i.e. the euclidean distance, as we are summing many distances.

# %%

result = {}

for num\_of\_clust in range(2, 10):

    clust, cent, total\_sum = cluster\_data(X, number\_of\_clusters=num\_of\_clust, show=False)

    result[num\_of\_clust] = total\_sum

fig, ax = plt.subplots(1, 1, figsize=(12, 8))

ax.plot(list(result.keys()), list(result.values()), 'bo--')

\_ = ax.set\_xlabel('Number of clusters (k)', fontsize=20)

\_ = ax.set\_ylabel('Loss', fontsize=20)

# %%

result.values()

# %% [markdown]

# The elbow of the graph is around 5, which is exactly the amount of clusters we defined in the beginning. Of course, this is a very nice toy dataset and with real data this might not be so clear. Still, nice to have a confirmation!

# %%

import os

os.getcwd()

# %% [markdown]

# # 2. Predict Titanic Survivabillity

# %% [markdown]

# We start with the dataset we have created during my KNN Notebook:

# %%

import pyarrow

import fastparquet

combined = pd.read\_parquet('titanic\_family\_survivabillity.parquet')

train = combined.loc[combined['set'] == 'train'].drop('set', axis=1).reset\_index(drop=True)

test = combined.loc[combined['set'] == 'test'].drop(['set', 'Survived'], axis=1).reset\_index(drop=True)

train['Survived'] = train['Survived'].astype(np.int)

columns = ['Pclass', 'Sex',  'Fare', 'family\_survival', 'family\_size']

# %%

clust, cent, total\_sum = cluster\_data(train[columns], number\_of\_clusters=100)

clust['Survived'] = train['Survived']

survival\_mapper = clust.groupby('cluster')['Survived'].mean().round().astype(np.int).to\_dict()

clust['group\_survived'] = clust.cluster.map(survival\_mapper)

clust

# %%

accuracy = (clust['Survived'] == clust['group\_survived']).mean()

accuracy  # train dataset accuracy

# %%

survival\_map = clust.groupby('cluster')['Survived'].mean().round().astype(np.int).to\_dict()

predict, loss = cluster\_dataset(test[columns], cent)

predict['Survived'] = predict.cluster.map(survival\_map)

predict['PassengerId'] = test['PassengerId']

predict = predict[['PassengerId', 'Survived']].sort\_values('PassengerId').reset\_index(drop=True)

predict

# %%

predict.to\_csv('results\_algorithm\_from\_scratch.csv', index=False)

# %% [markdown]

# On Kaggle, it scores 0.794. Not bad at all!

# %%

# %% [markdown]

# # 3. Solution using Scikit-Learn

# %%

from sklearn.cluster import KMeans

# %%

km = KMeans(

    n\_clusters=47,

    init='random',

    n\_init=32,

    max\_iter=600,

    tol=1e-5,

    random\_state=2020,

)

\_ = km.fit(train[['Pclass', 'Sex',  'Fare', 'family\_survival', 'family\_size']])

# %%

df = train.copy()

df['cluster'] = km.predict(df[columns])

mapper = df.groupby('cluster')['Survived'].mean().round().to\_dict()

df = test.copy()

df['Survived'] = km.predict(test[['Pclass', 'Sex',  'Fare', 'family\_survival', 'family\_size']])

df['Survived'] = df.Survived.map(mapper).astype(np.int)

df = df[['PassengerId', 'Survived']].sort\_values('PassengerId').reset\_index(drop=True)

df.to\_csv('results\_scikit\_algorithm.csv', index=False)

# %% [markdown]

# The Scikit-learn algorithm scores almost the same: 0.796

# %%