```
import numpy as np
import pandas as pd
from math import ceil
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

from pyspark.ml.clustering import KMeans, KMeansModel
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer, VectorIndexer
from pyspark.ml.feature import StandardScaler
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import PCA
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml.linalg import Vectors, SparseVector, DenseVector
from pyspark.ml import Pipeline

from pyspark.sql.types import *
from pyspark.sql.functions import col, round, sum, stddev_samp, when, udf, lit, substring, concat

"""
To create kmean model and evaluation
"""

class makeKMean:

    def __init__(self, df, feature_col, id_col='uid', scaling=False):
        """
        : param df: wifi log dataframe from parquet "wifi_fromto_yyyymmdd_yyyymmdd"
        (sessionized with from/to timestamps)
        : param feature_col: a list, features for the model
        : param id_col: a string, default to 'uid'
        :param scaling: default to False, whether or not to do feature scaling for continuous
        var (cont_num_col)
        """
        self.feature_col = feature_col
        self.id_col = id_col
        self.scaling = scaling
        self.num_col = ['num_ssid', 'num_user_name', 'num_user_type', 'num_device', 'num_prid']
        self.cat_col = ['campus', 'ssid', 'user_type', 'start_part_of_day']
        self.cont_num_col = ['session_length_minute', 'lag_length_minute', 'lag_ses_ratio']
        self.df_0 = df.select(*self.num_col, *self.cat_col, *self.cont_num_col, id_col).dropna()

        self.cont_num_scaled_col = [c+"_scaled" for c in self.cont_num_col]
        self.index_col = [c+"_index" for c in self.cat_col]
        self.vec_col = [c+"_vec" for c in self.cat_col]

        self.__features()
        self.__make_kmean_df()
        self.__make_sample()

    def __features(self):
        # normalizing "session_length" and "lag_length_minute", create the same result from
        using pyspark.ml.feature.StandardScaler (overkill for only two var)
        # also try min-max method
        # **Scaling data is usually necessary for distance-based algorithm. \
        # However, min-max and mean-sd methods are both sensitive to outliers, \
        # it may increase or decrease the effectiveness of outlier detection. \
        # More scaling methods should be explored!
        df = self.df_0
        if self.scaling:
            for c in self.cont_num_col:
                df = df.withColumn(c+'_scaled', col(c) / df.agg(stddev_samp(c)).first()[0])
```

```
        # transform categorical column into numerical indices (0, 1, 2...)
        indexers = [StringIndexer(inputCol=c, outputCol=c+"_index").fit(df) for c in
        self.cat_col]
        pipeline = Pipeline(stages=indexers)
        df = pipeline.fit(df).transform(df)

        # OneHotEncoderEstimator: translate categorical indices into several binary indicators
        # for example: indices 0,1,2 ==> vector of three features is_0, is_1, is_2
        encoder = OneHotEncoderEstimator(inputCols=self.index_col, outputCols=self.vec_col)
        df = encoder.fit(df).transform(df)
        self._df = df

    def __make_kmean_df(self):
        """
        vectorize features and return the index/features dataframe for kmean
        """
        num_col_fit = [c for c in self.feature_col if c in self.num_col]
        cat_col_fit = [c+'_vec' for c in self.feature_col if c in self.cat_col]
        if self.scaling:
            cont_num_col_fit = [c+'_scaled' for c in self.feature_col if c in self.cont_num_col]
        else:
            cont_num_col_fit = [c for c in self.feature_col if c in self.cont_num_col]
        kmean_df = self._df
        id_col = self.id_col
        kmean_df = kmean_df.select(*num_col_fit, *cat_col_fit, *cont_num_col_fit, id_col)
#         id_col_fit = kmean_df.columns[-1] # uid (last column)
        feature_col_fit = kmean_df.columns[:-1] # all other columns

        vecAssembler = VectorAssembler(inputCols=feature_col_fit, outputCol="features")
        self.kmean_df = vecAssembler.transform(kmean_df)

    def get_kmean_df(self):
        return self.kmean_df

    def get_model(self, k, seed=123):
        kmean_df = self.kmean_df
        kmeans = KMeans().setK(k).setSeed(seed).setFeaturesCol("features")
        model = kmeans.fit(kmean_df)
        return (kmeans, model)

    def get_prediction(self, model=None):
        model = self.get_model() if model is None else model
        kmean_df = self.kmean_df
        return model.transform(kmean_df)

    def get_cluster_distance(self, model=None, predict_df=None):
        model = self.get_model() if model is None else model
        predict_df = self.get_prediction() if predict_df is None else predict_df

        def cal_dist(vector, cluster):
            return float(np.linalg.norm(DenseVector(vector) - centers[cluster]))
        calDist = udf(cal_dist)

        centers = model.clusterCenters()
        predict_df = predict_df.withColumn('distance', calDist('features',
        'prediction').cast('double'))
        return self.df_0.join(predict_df.withColumnRenamed('prediction', 'cluster')\
                                        .select('uid', 'cluster', 'features', 'distance'), 'uid')

    def __make_sample(self, size=0.3):
        kmean_df = self.kmean_df
        self.sample_set = kmean_df.sample(False, size, seed=42)

    def sample_eval(self, n=20):
```

```
        sample_set = self.sample_set
        cost = np.zeros(n)
        silhouette = np.zeros(n)
        for k in range(2,n):
            kmeans = KMeans().setK(k).setSeed(1).setFeaturesCol("features")
            model = kmeans.fit(sample_set)
            cost[k] = model.computeCost(sample_set)
            #K-means cost = sum of squared distances of points to their nearest center.
            # Deprecated in 2.4.0. It will be removed in 3.0.0. Use ClusteringEvaluator instead

            transformed = model.transform(sample_set)
            evaluator = ClusteringEvaluator()
            silhouette[k] = evaluator.evaluate(transformed)
            #The metric computes the Silhouette measure using the squared Euclidean distance \
            #The Silhouette is a measure for the validation of the consistency within clusters. \
            #It ranges between 1 and -1, where a value close to 1 means that the points in a
            cluster \
            #are close to the other points in the same cluster and far from the points of the
            other cluster

        fig, ax = plt.subplots(1,1) #, figsize =(8,4)
        ax.plot(range(2,n),cost[2:n])
        plt.xticks(np.arange(2, n, 1));
        plt.grid(True);
        ax.set_xlabel('k');
        ax.set_ylabel('cost');
        plt.title('Sample K-mean Cost - '+', '.join(self.feature_col));
        plt.show()

        fig, ax = plt.subplots(1,1)
        ax.plot(range(2,n),silhouette[2:n])
        plt.xticks(np.arange(2, n, 1));
        plt.grid(True);
        ax.set_xlabel('k');
        ax.set_ylabel('squared euclidean distance');
        plt.title('Sample Silhouette Evalulation - '+', '.join(self.feature_col));
        plt.show()
```