CPSC 340 Machine Learning Take-Home Final Exam (Fall 2020)

1 Team

Team Members	*Jinlin Zhu(96057468;q6j8i), Xinyao Fan (72294374;x8x1b)*
Kaggle Team Name	*CPSC340*

2 Introduction (3 *points*)

The autonomous driving problem is basically building a model to predict the trajectory of a car of interest in the future 3 seconds given the past position information of both the ego car and other agents cars in the same intersection. The training set includes the coordinate information of ego car and the nine nearest other cars/pedestrians to the ego in the past one second before the prediction begins and also the future 3 seconds coordinate information of ego cars. The evaluation measure is RMSE of coordinates averaged over trajectories of 20 ego cars during 3 seconds.

3 Summary (12 *points*)

Each CSV file in the training/testing X represents the traffic of a intersection during one second. We found that the provided animation tools is less useful since too much un-related information or features from 'raw data' was involved in a mess in each frame. So we created our own animation solutions including only the data and features provided in the training set. Additionally, we inserted essential dynamics data such as velocity/acceleration vs time plot for the agent car in each trajectory files. With the help of our own animation videos, we visualized the trajectory of ego and other cars at each trajectory file 'frame by frame' in the training set. We noticed that some patterns or similarities of traffic among different trajectory files, by which we could select the models for use. Obviously, trajectories of ego cars in the test set are similar to some cases in the training set, which motivates us to use KNN model.

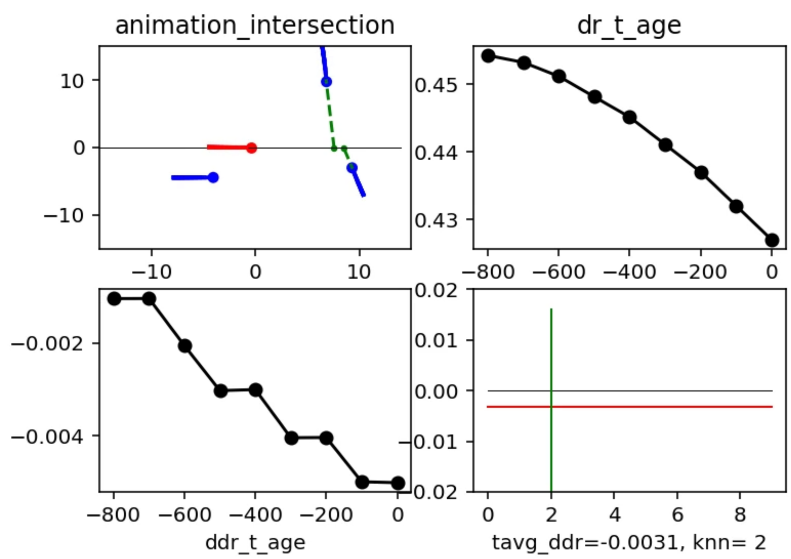
We developed two models based on KNN, by which we tried to interpret the similarity criterion of traffic trajectory files in different aspects. First model is called **Compare Trajectory of Agents KNN**, defined as CompareTrajAgeKNN class in code section. This is our primary model which only compares the similarity of trajectories of agent/ego cars among different trajectory files. Second model is called **Encounterability Analysis KNN**, defined as EncounterabilityKNN class in code section. This is a more advanced version which also tries to extract information from other 8 nearest neighbour cars in a single trajectory file. We analyzed the animations of trajectory files in details and created an analytical physical model by hand. We could see that only 0 or 1 nearest neighbour cars are correlated with the ego/agent car. Additionally, the dynamics data of the agent implied it only reacted to those neighbours of higher probabilities to block its way in following moments. More details on are provided in experiment section.

4 Experiments (15 *points*)

4.1 Data Prepossessing

We imported trajectory files from training set, validation set, test set into several lists with corresponding phase name such as X_train or X_val etc. Next, we filtered useless id/name information and extracted useful data into numpy arrays for further usage. Then we extracted the agent car coordinates to first two clumns and others' as the rest columns. All coordinates of non-existing cars with present=0 are filled with 'np.nan'. However, we noticed that quite a few trajectory files of y_train, y_val have less timesteps or rows than others. So we prolonged such time series up to 3 seconds with our **AutoRegression model** developed in midterm. Up to now, we have prepared the data in a unified format. The code dealing with this part are defined as read_data, data_prepare in code section. Further, the trajectory files are flattened and combined together as a trajectory matrix. So that each row of it represents a trajectory file with coordinates of each timesteps of each car as features. By feature engineering and our models, we could then utilize them with much convenience.

4.2 Data Visualization



We created our own animation tool that is defined as AnimatedScatter and its inherited AnimatedAnalysis class in code section. The above figure shows a screenshot of a frame of the animation. The top left subplot is a single frame of the trajectories of all nearest cars around the agent/ego which is generated from a single trajectory file of training sets. The green dash lines are the extension lines of trajectories of other cars in terms of their current velocity direction. This gives us indication that whether a neighbour is of higher probability to block the path of the agent. More details are described in Encounterability Model section. The top right and bottom left shows the velocity/acceleration data versus time for agent. The bottom right is a single moment correlation plot. It indicates how the agent car's reaction is correlated with its neighbours. The green line shows the number of possible important neighbours according to our model while its value is on the corresponding x-axis. The red line shows the averaged acceleration of the agent car, its value is on the corresponding y-axis. So if the agent car reacts to its neighbour cars, its acceleration will correlate to the number of neighbours. Thus the area enclosed by green-red lines would be large while their intersects would move on a straight line.

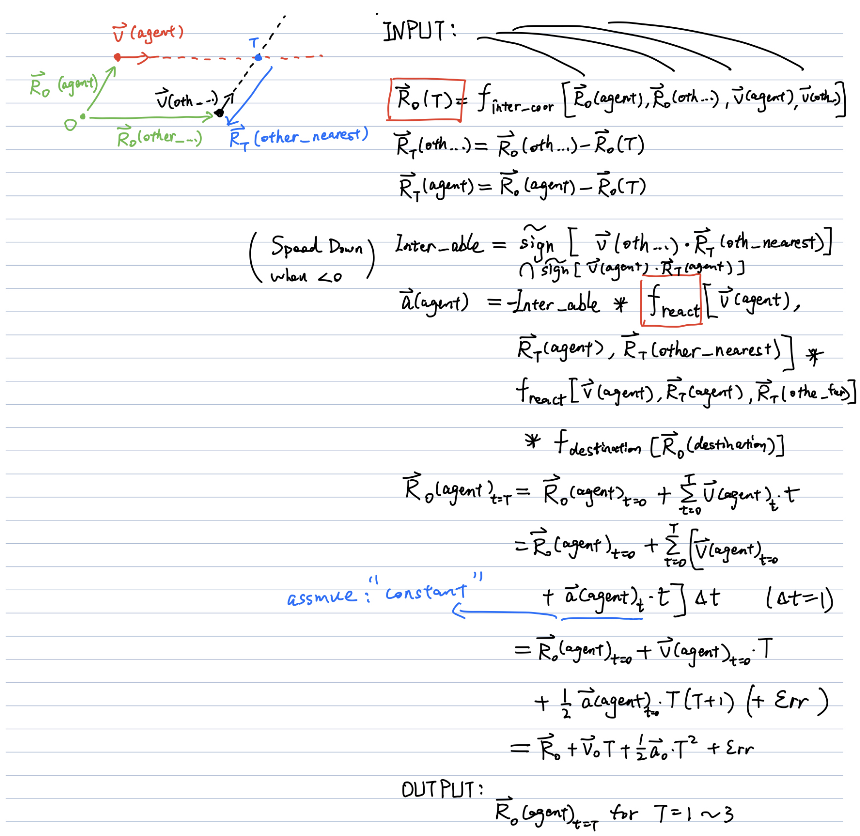
4.3 Feature Engineering

From the data visualization part, we noticed only less then two neighbours in a trajectory files may really affect the trajectory of the agent. So for each trajectory files, we only pick up to one neighbour for further use. The reason that we didn't pick two neighbours is that it will involve more noise to our model and it seems even one neighbour other than agent itself is too many. i.e. the more complicated model did nothing better. About how we select the neighbours in each single trajectory file, see Encounterability KNN model section.

4.4 CompareTrajAgeKNN Model

This is our primary model, however, it turns out that the simplest model works best. In this model, we only utilized the agent trajectory in each trajectory files. As we have said in data prepossessing, we flattened and combined all such trajectories as rows in a single matrix. So we have a trajectory matrix of training set and another of test set. Then we use our KNN model to select the k nearest neighbours from training matrix for each example of test matrix. We implemented cross-validation to select k, which is described in Hyper-parameter tuning section. Then using their distances to example of test matrix as weights, we averaged the corresponding y_train matrix and got final prediction of these text examples.

4.5 EncounterabilityKNN Model



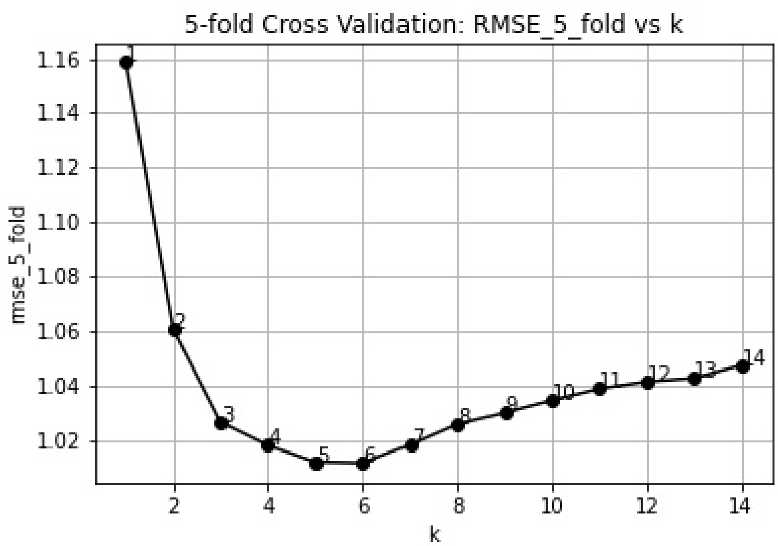
The above figure is part of the theory of encoutnerability model we derived by hand. We did this mainly by data visualization and dynamics data analysis. The main idea is that among nearest other cars/pedestrians, only those who has probability to block the path of agent will finally affect its acceleration and therefore affecting the trajectories. So we must filter out those un-related objects. According to our theory, we could draw extension lines such as we did in Data Visualization part for each neighbours of agent in a single trajectory file. Then the intersect between path of agent and the extension lines is called T point. We simply do the math in T-origin reference and get criterion of neighbour filtering.

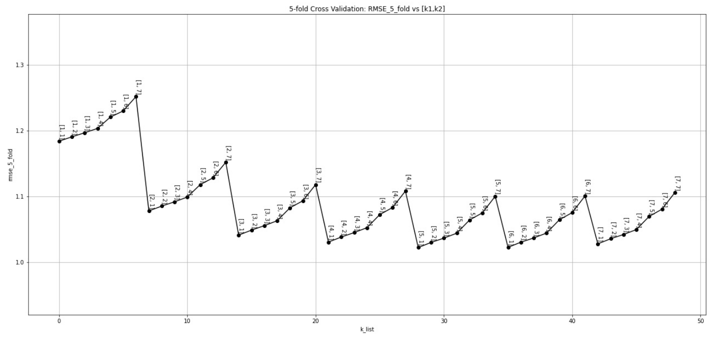
For this model, we divided both of the training and test set into two segments. For first segment of them, we only care about agent trajectory and no neighbour in each trajectory files since they are less correlated by our criterion. For second segment, we care about only one neighbour of the agent in each trajectory file. After all, for two segments, we implement KNN separately with two different k values, denoted as k_list where k1=k_list[0] for first KNN model and k2=k_list[1] for second KNN model. We will describe how we choose this k_list via cross validation in next Hyper-parameter Tuning section.

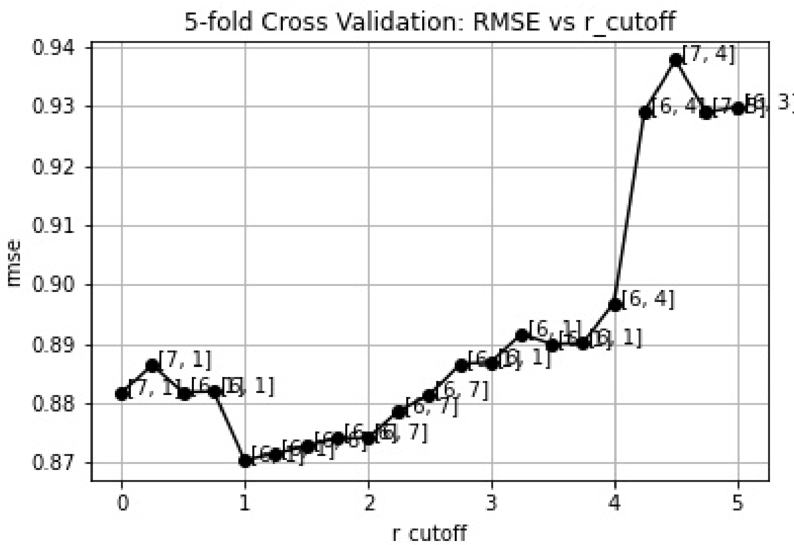
It is important to note that the EncounterabilityKNN model will shrink back to our primary CompareTrajAgeKNN model when second segment contains no trajectory files, so that only the agent trajectory was compared among different trajectory files. This happens when we set our encounterability(hyper-parameters that filter the un-related other cars/pedestrians) filter level too high.

Hyper-parameters for this model involves k_list, r_cutoff, which is the distance between the other car and the intersect point T. Some other hyper-parameters are fixed since they are tested as of less importance, this might be caused by some kind of optimization bias problem.

4.6 Hyper-paremeter Tuning







By 5-fold cross validation, for the primary model, k=6. For the advanced model, we measured our k_list with minimum rmse is [6,1] and r_cutoff is 1.

4.7 Evaluation

We created a baseline model which just did the average over all y_train and predict with this y_average. We compare all the models by validation set after we have tuned the hyper-parameters using cross-validation. During this validation phase, the baseline model gives a rmse as 2.624 while our primary model gives a minimum rmse of 0.882 and our second advanced model gives a minimum rmse of 0.870. However, we noticed that when it comes to the final test set, our advanced model shrinks back to primary model because some patterns of traffic is not contained in test set though it appears in validation set.

5 Results (5 *points*)

Team Name	Kaggle Score
CPSC340	*0.54060*

6 Conclusion (5 *points*)

In this project, we have learned how to create animation by python and analyze data with it. We tried to solve the problem by primary KNN model but we also tried to improve it in all perspectives. The final project contains thousands of lines of codes and we debugged it again and again until all going smooth. Though our advanced model didn't seem to improve the performance much but it consumed significantly more energy and time than its primary version. The codes themselves saw what we have learnt so far. However, there is still a lot for us to do if we were given more time, such as change the prototype model from KNN to like PCA, by which we could compare the similarity between each trajectory file by matrix factorization and so forth. Further, we could have made an ensemble method to select the best predictions overall.

7 Code

```


from utils import sign_able
from utils import get_rmse
from utils import len_notnull
from utils import dynamics_tavg_data
from utils import dynamics_time_data_t0
from utils import Coor_T_xy_array
import numpy as np
from knn import KNN















class CompareTrajAgeKNN():
    def __init__(self,X_train,y_train,k):
        '''
        Parameters
        ----------
        X_train : Dataset type form read_data
        y_train : Dataset type form read_data
        k : int
            number of nearest neighbours (KNN)
        Stores Datasets and initializes submodel with hyperparameter k
        -------
        '''
        self.X_train=X_train
        self.y_train=y_train
        self.k=k
        self.submodel = KNN(k=k)




    def fit(self,X2test):
        '''
        Parameters
        ----------
```

```
        X2test : Dataset type form read_data
            has to be processed using 'data_preapare' and 'subsets2trajs'

        Stores processed trajectory matrix satisfying interface of submodel (KNN)
        Turns in the matrix to be used into submodel (KNN)
        -------
        '''

        self.feature_selcetion(self.X_train,type_phase='train',method='knn')
        self.X_train_trajs = self.subsets2trajs(self.X_train)
        self.y_train_trajs = self.subsets2trajs(self.y_train,type_dataset='y')

        self.feature_selcetion(X2test,type_phase='test',method='knn')
        self.X2test_trajs = self.subsets2trajs(X2test,type_phase='test')

        self.submodel.fit(self.X_train_trajs,self.y_train_trajs)

    def predict(self):
        '''
        Predict using submodel (KNN) form stored usable matrix
        Returns y_pred with original format: time series
        '''
        y_pred_trajs = self.submodel.predict(self.X2test_trajs)
        #y_pred = self.traj2matrix(y_pred)
        return y_pred_trajs

    def val_phase(self,X_val,y_val):
        '''
        Input validation sets,
        fit and predict using the self model
        returns rmse, flattened y_pred and corresponding y_val
        '''
        self.fit(X_val)
        y_pred_trajs = self.predict()
        y_val_trajs = self.subsets2trajs(y_val,type_dataset='y',type_phase='test')
        rmse = get_rmse(y_pred_trajs,y_val_trajs,type_obj='traj')
        return rmse,y_pred_trajs,y_val_trajs

    def feature_selcetion(self,dataset,type_phase='train',method='knn'):
        '''
        method == 'knn':
            Select the features of each traj file, using find_knn_subsets function
            Then all ids of selected dim are stored in lists: inds_knn_subset, inds_subsets_knn_classes
        phase_type=='train':
            Filters and stores ids of features of training set , otherwise of test set
        '''
        self.type_phase=type_phase
        if method=='defualt':

            self.knn_classes=np.array([9],dtype=np.int32)

            inds_defult=np.array([0,1,2,3,4,5,6,7,8])
            inds_defult=np.repeat(inds_defult[:,None],len(self.X_train)-1,axis=1).T

            #store the indices of its knn regarding the subset index
            self.inds_knn_subset=inds_defult

            #store the indices of subsets regarding the knn class number
            self.inds_subsets_knn_classes=[[]]
            self.inds_subsets_knn_classes[0].append(list(range(len(self.X_train))))
        if method=='knn':
            self.knn_classes=[0] #0 means 0 neighbor near agent, 1 means 1 , so on and so forth

```

```python
            if type_phase=='train':

                self.inds_subsets_knn_classes,self.inds_knn_subset = self.find_knn_subsets(dataset)

            if type_phase=='test':
                self.test_inds_subsets_knn_classes,self.test_inds_knn_subset = self.find_knn_subsets(dataset)


    def find_knn_subsets(self,dataset):
        '''
        ----------
        Variables:

        inds_knn_subset : N by k list, dtype= int, where k is k_knn (0 means only agent, 1 means 1 neighbor)
            Stores the id of sorted k nearest neighbours (including agent as id=0)

        inds_subsets_knn_classes: len(knn_classes) by * list, dtype= int
            Stores the id of subsets with corresponding number of nearest neighbours

            eg:
            for training phase, inds_subsets_knn_classes[1]=[2,3]
            means in subsets X_train[2], X_train[3] only two cars including agent are considered correlated,
            thus being classified to the id=1 in knn_class=[0,1]

        Description:
            using KNN to find the k neareset neighbours of the first row (agent) from the traj matrix within the traj file,
            where traj matrix is composed by trajs of all cars in a single traj file
            inds filter with more strict rule could be added after the implementation of KNN model
            then ids of k nearest neighbours for each traj file (like X_train[i]) are stored
            and ids of subsets are classified according to the (filtered) number of knn

        Returns inds_subsets_knn_classes, inds_knn_subset
        -------

        '''

        #initialize
        inds_knn_subset=[]
        inds_subsets_knn_classes=[[]]

        for i in range(len(dataset)):
            model = KNN(k=max(self.knn_classes)+1)
            subset_trajs_mat=self.subsets2trajs(dataset[i],type_dataset='subset')

            inds_knn,_=model.find_inds_knn(subset_trajs_mat[0:],subset_trajs_mat[0][None])
            inds_knn_list=inds_knn.flatten().tolist()
            # inds_knn_list=[x+1 for x in inds_knn_list]
            # inds_knn_list_age=[0]+inds_knn_list


            #add inds filter here
            #replace next inds_knn_subset[i] by new indices
            #then the filtered k might be smaller than the input k at the init
            #if len(inds_knn.flatten()) <= max(self.knn_classes): #ignore?

            inds_knn_subset.append(inds_knn_list)


            inds_subsets_knn_classes[len(inds_knn_list)-1].append(i) ###??????????????????
```

```
        return inds_subsets_knn_classes, inds_knn_subset


    def subset_selection(self,subset,id_sel,axis=0):
        '''
        Parameters
        ----------
        dataset : list
            Dataset like X_train after read_data()

        ind_subset : int
            Index of selected subset from Dataset

        id_sel: 1D numpy array of int
            IDs of car selected (or timesteps if axis=1)

        axis : 1: select the whole row regarding indices as of rows
               0: select the whole column regarding indices as of columns

            DESCRIPTION. The default is 1.

        Returns
        -------
        subset_sel : single subset from Dataset with selected columns of id_sel

        '''

        if axis == 0:#defualt select whole columns
            inds_dim_xy = self.id2xy(id_sel)
            subset_sel=subset[:,inds_dim_xy]
        elif axis == 1:
            inds_dim_xy = self.id2xy(id_sel)
            subset_sel=subset[inds_dim_xy]
        return subset_sel


    def id2xy(self,id_cars):
        '''
        Input: virtual id(not the id of cars) of others or agent(id=0) in order of the prepared subset

        Returns real indices of xy in subset from Dataset
        -------

        '''
        if type(id_cars)==list:
            id_cars=np.asarray(id_cars,dtype=np.int32)
        ind_x=id_cars*2
        ind_y=id_cars*2+1

        if type(id_cars) is int :
            ind_xy=np.array([0,1])
            ind_xy[0]=ind_x
            ind_xy[1]=ind_y
        else:
            ind_xy=np.concatenate((ind_x[:,None],ind_y[:,None]),axis=1).flatten()

        return ind_xy


    def traj2matrix(self,traj):
        '''
        Input flattened subsets
        Returns original subsets
        '''
```

```

        raise NotImplementedError # TODO

    def subsets2trajs(self,dataset,knn_class=0,type_dataset='X',type_phase='train'):
        '''
        Parameter:

        dataset: list
            list of traj files, such as X_train or y_train

        knn_class: int
            only required for 'X' dataset_type
            knn_class=0 means only agent is considered, 1 means 1 nearset neighbor class, so on and so forth.

        type_dataset: string
            marks the type of dataset, X means input matrix, y means output matrix, subset means X[i] which is the traj file

        self.type_phase: string
            marks the phase of making the matrix, 'train' means training phase, 'test' means test phase
            'train phase' makes adjacent traj matrix for training set
            while during 'test phase' makes adjacent traj matrix for test set

        Describtion:
            Create the matrix of trajectories as rows, which statisfies the interface of original KNN
            So that for each row represents flattened traf from X_train[i] or y_train[i], with num_cars*2*timesteps being dims

            First dim of trajs_mat is the length of X_train

        Returns trajs_mat
        --------
        '''
        len_dataset=len(dataset)



        if type_dataset=='subset':


            num_cars=int(len_notnull(dataset)/2)
            trajs_mat = np.zeros((num_cars,len_dataset*2))
            for i in range(num_cars):
                car_xyt =self.subset_selection(dataset,i)

                trajs_mat[i] = car_xyt.flatten()

        elif type_dataset=='X' and type_phase=='train':
            inds_subset = self.inds_subsets_knn_classes[knn_class]
            num_subsets=len(inds_subset)
            trajs_mat = np.zeros((num_subsets,len(dataset[0])*(knn_class+1)*2))
            for i in range(num_subsets):
                ind_subset=inds_subset[i]

                subset=dataset[ind_subset]

                subset_sel = self.subset_selection(subset,self.inds_knn_subset[ind_subset]) #inds_subset includes 0 as agent for each row

                trajs_mat[i] = subset_sel.flatten()

        elif type_dataset=='X' and type_phase=='test':
```

```python
            inds_subset = self.test_inds_subsets_knn_classes[knn_class]
            num_subsets=len(inds_subset)
            trajs_mat = np.zeros((num_subsets,len(dataset[0])*(knn_class+1)*2))
            for i in range(num_subsets):
                ind_subset=inds_subset[i]

                subset=dataset[ind_subset]

                subset_sel = self.subset_selection(subset,self.test_inds_knn_subset[ind_subset]) #inds_subset includes 0 as agent for each row

                trajs_mat[i] = subset_sel.flatten()

        elif type_dataset=='y' and type_phase=='train':
            inds_subset = self.inds_subsets_knn_classes[knn_class]
            num_subsets=len(inds_subset)
            trajs_mat = np.zeros((num_subsets,len(dataset[0])*2))
            for i in range(num_subsets):
                ind_subset=inds_subset[i]
                subset_sel = dataset[ind_subset]

                trajs_mat[i] = subset_sel.flatten()

        elif type_dataset=='y' and type_phase=='test':

            trajs_mat = np.zeros((self.len_X2test,len(dataset[0])*2))
            for i in range(self.len_X2test):

                subset_sel = dataset[i]

                trajs_mat[i] = subset_sel.flatten()

        return trajs_mat



class EncounterabilityKNN(CompareTrajAgeKNN):
    def __init__(self,X_train,y_train,k_list=[6,6]):

        self.X_train=X_train
        self.y_train=y_train
        #k is the k_knn of similar trajfiles regarding the test trajfile
        #type of classes of different k-knn, 1 means agent itself, 2 means 1 NN


        self.k_list=k_list
        self.submodels=[]
        self.submodels.append(KNN(k=self.k_list[0]))
        self.submodels.append(KNN(k=self.k_list[1]))


        self.Hparameter_init()
        #initialize submodels (KNN in terms of multiple traj knn_classes)



    def Hparameter_init(self,filter_lv='strict_time',
                        r_T_cutoff=10,time_cutoff=10,HalfCarLength_PassingTime=0.5,
                        knn_classes=2,k_traj_knn=10):
        self.filter_lv=filter_lv

        self.r_T_cutoff=r_T_cutoff
        self.time_cutoff=time_cutoff
        self.HalfCarLength_PassingTime=HalfCarLength_PassingTime

```

```python
        self.k_traj_knn=k_traj_knn
        self.knn_classes=knn_classes

        if knn_classes:
            self.knn_classes=knn_classes
        if k_traj_knn:
            self.k_traj_knn=k_traj_knn




        # self.k_list =np.array([self.k0,self.k1],dtype=np.int8)

        # # if not k_list:
        # #     self.k_list=np.ones(self.knn_classes,dtype=np.int8)*self.k
        # #     print("K_list arranged as [k,k]")

        # if len(self.k_list) != self.knn_classes:
        #     print("Error: k_list input form incorrect!")
        #     return

    def fit(self,X2test):
        '''
        Parameters
        ----------
        X2test : Dataset type form read_data
            has to be processed using 'data_preapare' and 'subsets2trajs'

        Stores processed trajectory matrix satisfying interface of submodel (KNN)
        Turns in the matrix to be used into submodel (KNN)
        -------
        '''

        self.len_X2test=len(X2test)

        self.feature_selcetion(self.X_train,type_phase='train')
        self.feature_selcetion(X2test,type_phase='test')
        self.X_train_trajs = []
        self.y_train_trajs = []
        self.X2test_trajs = []


        for i in range(self.knn_classes):

            self.X_train_trajs.append(self.subsets2trajs(self.X_train,knn_class=i,type_phase='train'))
            self.y_train_trajs.append(self.subsets2trajs(self.y_train,knn_class=i,type_dataset='y',type_phase='train'))#???
            self.X2test_trajs.append(self.subsets2trajs(X2test,knn_class=i,type_phase='test'))

            self.submodels[i].fit(self.X_train_trajs[i],self.y_train_trajs[i])#???
            self.check_sanity(self.X_train_trajs[i],self.inds_subsets_knn_classes[i],i)
            self.check_sanity(self.y_train_trajs[i],self.inds_subsets_knn_classes[i],i)
            self.check_sanity(self.X2test_trajs[i],self.test_inds_subsets_knn_classes[i],i)

    def check_sanity(self,A,B,c=1):
        if len(A)!=len(B):
            print("Error: Class %d: lens between A(%d) and B(%d) mismached."%(c,len(A),len(B)))

    def predict(self):
        '''
        Predict using submodel (KNN) form stored usable matrix
        Returns y_pred with original format: time series
```

```
        '''
        y_pred_trajs=[]
        y_pred_trajs_final = np.zeros((self.len_X2test,self.y_train_trajs[0].shape[1]))

        for i in range(self.knn_classes):
            y_pred_trajs.append(self.submodels[i].predict(self.X2test_trajs[i]))
            y_pred_trajs_final[self.test_inds_subsets_knn_classes[i]]=y_pred_trajs[i]

        return y_pred_trajs_final


    def feature_selcetion(self,dataset,type_phase='train'):
        '''
        method == 'knn':
            Select the features of each traj file, using find_knn_subsets function
            Then all ids of selected dim are stored in lists: inds_knn_subset, inds_subsets_knn_classes
        phase_type=='train':
            Filters and stores ids of features of training set , otherwise of test set
        '''
         #0 means 0 neighbor near agent, 1 means 1 , so on and so forth

        if type_phase=='train':

            self.inds_subsets_knn_classes,self.inds_knn_subset = self.find_knn_subsets(dataset)

        if type_phase=='test':
            self.test_inds_subsets_knn_classes,self.test_inds_knn_subset = self.find_knn_subsets(dataset)




    def find_knn_subsets(self,dataset):

        #initialize
        inds_knn_subset=[]
        inds_subsets_knn_classes=[]
        for i in range(self.knn_classes):
            inds_subsets_knn_classes.append([])

        for i in range(len(dataset)):
            model = KNN(k=self.k_traj_knn)
            subset_trajs_mat=self.subsets2trajs(dataset[i],type_dataset='subset')

            inds_knn,_=model.find_inds_knn(subset_trajs_mat[0:],subset_trajs_mat[0][None])
            inds_knn_flat=inds_knn.flatten()


            #add inds filter here

            inds_knn=self.inds_knn_filter(dataset[i],inds_knn_flat[1:])


            #replace next inds_knn_subset[i] by new indices
            #then the filtered k might be smaller than the input k at the init
            inds_knn_list=[0]+inds_knn.tolist()

            if len(inds_knn_list) > self.knn_classes:
                inds_knn_list=inds_knn_list[:self.knn_classes]

            inds_knn_subset.append(inds_knn_list)
            inds_subsets_knn_classes[len(inds_knn_list)-1].append(i)
```

```python

        return inds_subsets_knn_classes, inds_knn_subset


    def inds_knn_filter(self,trajfile,ind_knn):
        xy_T,t_age,t_oth,dot_age,dot_oth,r_age_T,r_oth_T = self.Coor_T_knn_tavg(trajfile,ind_knn)

        time_cutoff = self.time_cutoff
        r_T_cutoff = self.r_T_cutoff
        sign_dot_age=sign_able(dot_age)
        sign_dot_oth=sign_able(dot_oth)
        if self.filter_lv == 'loose':
            sign_final = sign_dot_age * sign_dot_oth
        elif self.filter_lv == 'strict_time':
            sign_time_dif = sign_able(t_oth-t_age)
            sign_time_age = sign_able(t_age-time_cutoff)
            sign_time_oth = sign_able(t_oth-time_cutoff)
            sign_final = sign_dot_age * sign_dot_oth * sign_time_dif * sign_time_age * sign_time_oth
        elif self.filter_lv == 'strict_space':
            sign_space_age = sign_able(r_age_T - r_T_cutoff)
            sign_space_oth = sign_able(r_oth_T - r_T_cutoff)
            sign_final = sign_dot_age * sign_dot_oth * sign_space_age * sign_space_oth
        elif self.filter_lv == 'strict_hybrid':
            sign_time_oth = sign_able(t_oth-time_cutoff)
            sign_space_age = sign_able(r_age_T - r_T_cutoff)
            sign_space_oth = sign_able(r_oth_T - r_T_cutoff)
            sign_final = sign_dot_age * sign_dot_oth * sign_space_age * sign_space_oth * sign_time_oth
        if len(sign_final) != len(ind_knn):
            print("Error: size of signs and ind_knn not match!")
        ind_knn_new = ind_knn[sign_final!=0]
        return ind_knn_new

    def Coor_T_knn_tavg(self,trajfile,ind_knn):
        ind_age=0
        le = len(ind_knn)
        xy_age, dxy_age, ddxy_age, r_age, dr_age, ddr_age = np.zeros((le,2)),np.zeros((le,2)),np.zeros((le,2)),np.zeros(le),np.zeros(le),np.zeros(le)
        xy_oth, dxy_oth, ddxy_oth, r_oth, dr_oth, ddr_oth = np.zeros((le,2)),np.zeros((le,2)),np.zeros((le,2)),np.zeros(le),np.zeros(le),np.zeros(le)
        xy_age0, dxy_age0, ddxy_age0, r_age0, dr_age0, ddr_age0 = dynamics_time_data_t0(trajfile,ind_age)
        for ind in range(le):
           xy_age[ind], dxy_age[ind], ddxy_age[ind], r_age[ind], dr_age[ind], ddr_age[ind] = xy_age0, dxy_age0, ddxy_age0, r_age0, dr_age0, ddr_age0
           xy_oth[ind], dxy_oth[ind], ddxy_oth[ind], r_oth[ind], dr_oth[ind], ddr_oth[ind] = dynamics_tavg_data(trajfile,ind_knn[ind])


        x_T,y_T = Coor_T_xy_array(xy_age,xy_oth,dxy_age,dxy_oth)
        xy_T = np.concatenate((x_T[:,None],y_T[:,None]),axis=1)

        xy_age_T = xy_age-xy_T
        xy_oth_T = xy_oth-xy_T
        xy_oth_tail_T = xy_oth-xy_T - self.HalfCarLength_PassingTime * dxy_oth

        r_age_T = np.linalg.norm(xy_age_T,axis=1)
        r_oth_T = np.linalg.norm(xy_oth_T,axis=1)

        t_age = r_age_T/dr_age
        t_oth = r_oth_T/dr_oth

        dot_age = np.diag(xy_age_T@dxy_age.T)
```

```
        dot_oth = np.diag(xy_oth_tail_T@dxy_oth.T)

        return xy_T,t_age,t_oth,dot_age,dot_oth,r_age_T,r_oth_T

#if input <0 output 1; input >0 output 0; input = 0 output 0.5
```

```
import os
import utils
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.animation import FuncAnimation
from models import EncounterabilityKNN


class AnimatedScatter(object):
    def __init__(self,I,scale=30,num=0,save=True):
        self.scale = scale
        self.set_up_init(I)
        self.anim = FuncAnimation(self.fig, self.update, fargs=(I,), frames=np.arange(11), interval=200)
        self.save_fig(num, save)

    def set_up_init(self,I):
        self.fig, self.ax = plt.subplots()
        self.num_cars = int(utils.len_notnull(I)/2)

        self.le = self.num_cars*2

        self.age_scat = self.ax.scatter([I[0,0]],[I[0,1]],c='r')
        self.oth_scat = self.ax.scatter([I[0,2:self.le:2]],[I[0,3:self.le:2]],c='b')



        self.trajs =[None]*self.num_cars
        for i in range(self.num_cars):
            self.trajs[i] = self.ax.plot(I[0,2*i],I[0,2*i+1],lw=2)

        #self.ax.axis([np.min(I[:,0::2]), np.max(I[:,0::2]), np.min(I[:,1::2]), np.max(I[:,1::2])])
        scale=self.scale
        self.ax.axis([-scale,scale,-scale,scale])
        self.ax.set_autoscale_on(False)

    def update(self, t,I):

        temp = np.zeros((self.num_cars-1,2))
        temp[:,0] =I[t,2:self.le:2]
        temp[:,1] =I[t,3:self.le:2]
        self.age_scat.set_offsets((I[t,0],I[t,1]))
        self.oth_scat.set_offsets(temp)

        self.trajs[0] = self.ax.plot(I[0:t,0],I[0:t,1],'r-',lw=2)
        for i in range(1,self.num_cars):
            self.trajs[i] = self.ax.plot(I[0:t,2*i],I[0:t,2*i+1],'b-',lw=2)

        dr,dtan = utils.dynamics_data(I,t)


        label = 'timestep {0}, '.format(-1000+t*100)+'age_dr= %.3f, '%dr + 'age_dtan= %.3f, '%dtan
        self.ax.set_xlabel(label)
        return self.ax

    def save_fig(self,num,save):
```

```python
        if save == True:
            plotname='Intersection_'+str(num)+'.mp4'
            fname = os.path.join("..",'figs',plotname)
            self.anim.save(fname, dpi=144, fps=2, extra_args=['-vcodec', 'libx264'])
        else:
            plt.show()
        plt.cla()
        plt.clf()
        plt.close('all')


class AnimatedAnalysis(AnimatedScatter):
    def __init__(self,I,ind_knn,scale=30,num=0,save=True,anim_label=False):
        self.I = I
        self.scale = scale
        self.ind_knn=ind_knn
        self.anim_label=anim_label

        self.set_up_init()
        self.anim = FuncAnimation(self.fig, self.update, fargs=(I,), frames=np.arange(11),
    interval=200)
        self.save_fig(num, save)


    def set_up_init(self):
        self.fig, ((self.ax1,self.ax2),(self.ax3,self.ax4)) = plt.subplots(2,2)
        self.num_cars = int(utils.len_notnull(self.I)/2)
        self.le = self.num_cars*2

        self.draw_points_init()
        self.draw_data_init()
        self.draw_traj_init()
        self.filtered_proj_init()

    def draw_points_init(self):
        #I_knn=utils.I_inds(self.I,self.ind_knn)
        age_points_size=20
        oth_points_size=20
        T_points_size=5

        self.ax1.set_title("animation_intersection")
        self.age_scat = self.ax1.scatter([self.I[0,0]],[self.I[0,1]],c='r')
        age_sizes=np.ones(1)
        self.age_scat.set_sizes(age_sizes*age_points_size)

        if self.num_cars >1:
            self.oth_scat = self.ax1.scatter([self.I[0,2:self.le:2]],[self.I[0,3:self.le
    :2]],c='b')
            oth_sizes=np.ones(self.num_cars-1)
            self.oth_scat.set_sizes(oth_sizes*oth_points_size)


        self.xy_T=EncounterabilityKNN(self.I,self.I).Coor_T_knn_tavg(self.I,self.ind_knn)[0]
        T_sizes=np.ones(self.xy_T.shape[0])
        self.T = self.ax1.scatter(self.xy_T[:,0],self.xy_T[:,1],c='g')
        self.T.set_sizes(T_sizes*T_points_size)

        scale=self.scale
        self.ax1.axis([-scale,scale,-scale,scale])
        self.ax1.set_autoscale_on(False)

    def draw_data_init(self):
        r_t_age,dr_t_age,ddr_t_age=utils.dynamics_time_data(self.I,0)[3:]
        self.plot_xt(self.ax2,dr_t_age)
        self.ax2.set_title("dr_t_age")
```

```
        self.plot_xt(self.ax3,ddr_t_age)
        self.ax3.set_xlabel("ddr_t_age")

        #self.plot_xt(self.ax4,tavg_ddr)

        #Plot the final data analaysis plot in terms of statstical information
        num_knn=len(self.ind_knn)
        tavg_ddr = ddr_t_age.mean()

        #Range functions
        range_num=np.arange(0,10,1)
        ymin=-0.02
        ymax=0.02
        range_y=np.arange(ymin,ymax,(ymax-ymin)/10)
        ones=np.ones(len(range_num))

        self.ax4.plot(range_num,ones*0,'k-',lw=0.5)
        self.ax4.plot(range_num,ones*tavg_ddr,'r-',lw=1)
        self.ax4.plot(ones*num_knn,range_y,'g-',lw=1)
        #self.ax4.plot([num_knn],[tavg_ddr],'bo')


        self.ax4.set_ylim(ymin,ymax)
        self.ax4.set_xlabel("tavg_ddr=%.4f, knn= %d"%(tavg_ddr,len(self.ind_knn)))

    def draw_traj_init(self):
        self.trajs =[None]*self.num_cars
        for i in range(self.num_cars):
            self.trajs[i] = self.ax1.plot(self.I[0,2*i],self.I[0,2*i+1],lw=2)

    def filtered_proj_init(self):
        # ind_oth = ind_knn_filter(self.I,self.ind_knn)
        # xy_T_new = Coor_T_knn_tavg(self.I,ind_oth)[0]
        # I_new = utils.I_inds(self.I,ind_oth)
        # I_oth =I_new[:,2:]
        I_oth = utils.I_inds(self.I,self.ind_knn)
        for i in range(len(self.ind_knn)):
            x_T= self.xy_T[i,0]
            y_T= self.xy_T[i,1]
            x_oth = I_oth[0,2*i]
            y_oth = I_oth[0,2*i+1]
            # print(np.array([x_T,x_oth]))
            # print(np.array([y_T,y_oth]))
            self.projs =self.ax1.plot(np.array([x_T,x_oth]),np.array([y_T,y_oth]),'g--')
        scale=self.scale
        self.ax1.plot(np.arange(-scale,scale,1),np.arange(-scale,scale,1)*0,'k-',lw=0.5)

    def update(self, t,I):

        #Update the location of agent
        self.age_scat.set_offsets((I[t,0],I[t,1]))
        #Update the trajectories of cars.
        self.trajs[0] = self.ax1.plot(I[0:t,0],I[0:t,1],'r-',lw=2)
        if self.num_cars >1:
            #Update the location of other cars
            temp = np.zeros((self.num_cars-1,2))
            temp[:,0] =I[t,2:self.le:2]
            temp[:,1] =I[t,3:self.le:2]
            self.oth_scat.set_offsets(temp)
        for i in range(1,self.num_cars):
            self.trajs[i] = self.ax1.plot(I[0:t,2*i],I[0:t,2*i+1],'b-',lw=2)

        dr,dtan = utils.dynamics_data(I,t)

        if self.anim_label:
```

```python
            label = 'timestep {0}, '.format(-1000+t*100)+'age_dr= %.3f, '%dr + 'age_dtan= %.3f, '%dtan
            self.ax1.set_xlabel(label)

        return self.ax1

    def plot_xt(self,ax,x):
        ax.plot(utils.range_t(x),x,'ko-')
```

```python


import utils
import numpy as np
import pandas as pd
from scipy import stats


def get_similarity(coor_A,coor_B):
    """
    #find shortest possible coordinate vector min_coor_B in terms of coor_A and return the Distance
    #where min_coor_B is reorganized according to one of the permutaion of [1,2,3,...k] and coor_B
    #eg. ind_B= [3,5,6,1,2,7,...] (10 by 1) use this indice sequence to reoganize coor_B so that coor_B = [x3,y3,x5,y5,x6,y6,x1,y1,...] (20 by 1)
    #find the sequence that returns shortest distance score of Distance Matrix of A,B

    #coor_A is a 20 by 1 vector, containing coordinates of car 0-9 from test set
    #coor_B is a 20 by 1 vector, containing coordinates of car 0-9 from training set
    #min_ind_B is a 10 by 1 vector, containing indices of indices of reorganized B, providing shortest distance to A
    #min_coor_B is a 20 by 1 vector, containing coordinates of reorganized B, providing shortest distance to A
    #Dist is the global shortest distance among all possible pair
    #Dist is calculated by summing over all elements from a Distance Matrix which is computed by coor_A and min_coor_B

    #Procedure:
    #1. find next permutation of B
    #2. get Distance_Matrix_temp
    #3. get Dist_temp
    #4. compare Dist_temp vs Dist_min
    #5. update Dist_min #, min_ind_B, min_coor_B
    #6. if this is not the last permutation case, go to 1
    Dist = 0
    raise NotImplementedError # TODO
    return Dist #optional: also return min_ind_B, min_coor_B

    #note that Cost = O(k!*k(k-1)/2)
    #where k is half the lenth of coor_A or coor_B
    #so Cost ~ 163,296,000 times Cost[dist((x0,y0),(x1,y1))]
    """




def knn_frame(I,t,k):
    '''Find k nearest neighbor for the agent car, in frame of timestep = f(t) = -1000+100*t, t in (0,...10)

    Input: dataframe of a trajectory file with certain time index t =num_rows

    Return: the indices of knn and their distances towards the agent
            ind_knn, dist2_knn are k by 1, 1D arrays
    '''
```

```
    le = utils.len_notnull(I)
    frame_coor = I[t][:le]
    pairwise_coor = np.concatenate((frame_coor[::2][:,None],frame_coor[1::2][:,None]),axis=1)
    coor_tar = pairwise_coor[0][None]
    dist2 = utils.DistanceMatrix(pairwise_coor[1:],coor_tar).euclidean_dist_squared()
    ind_knn = np.argsort(dist2,axis=0)[:k]
    dist2_knn = np.zeros(k)
    dist2_knn = dist2[ind_knn]

    return ind_knn.flatten()+1, dist2_knn.flatten()



# def sign_inter_able(dot_age,dot_oth,t_age,t_oth,r_age_T,r_oth_T,strict_lv='strict_timing',time_cutoff=10,r_T_cutoff=50):

#      return sign_final









class KNN:

    def __init__(self, k):
        self.k = k

    def fit(self, X, y):
        self.X= X # just memorize the trianing data
        self.y = y
        self.k = min(self.k, self.X.shape[0])

    def predict(self, X_test,method='weight'):
        # Compute distance distances between X and Xtest


        y_pred = np.zeros((X_test.shape[0],self.y.shape[1]))

        self.find_inds_knn(self.X,X_test)
        if method == 'mean':
            y_pred=np.mean(self.y[self.inds_knn],axis=1)
        # print(self.y.shape)
        # print(y_pred.shape)
        # print(self.inds_knn.shape)
        # print(self.dist2_knn.shape)
        # print(self.y[self.inds_knn].shape)
        if method == 'weight':
            weights=self.nomalize_dist(self.dist2_knn)
            y_weighted=weights[:,:,None]*self.y[self.inds_knn]
            y_pred=np.sum(y_weighted,axis=1)

        return y_pred

    def find_inds_knn(self,X, X_test):
        dist2 = utils.DistanceMatrix(X,X_test).euclidean_dist_squared()
        self.inds_knn = np.argsort(dist2,axis=0)[:self.k].T
        self.dist2_knn = dist2[self.inds_knn][0].T #???
        #print(self.dist2_knn)
        return self.inds_knn,self.dist2_knn

    def nomalize_dist(self, dist2):
```

```
        sum_dist2=np.sum(dist2,axis=1)
        res= dist2/sum_dist2[:,None]
        return res
```

```

import os
import pandas as pd
import numpy as np

from sklearn.utils.validation import check_array



class DistanceMatrix():
    def __init__(self,A,B):
        self.A=A
        self.B=B

    def check_vec2matrix(self):
        if self.A.ndim==1:
            self.A=self.A[None]
        if self.B.ndim==1:
            self.B=self.B[None]

    def check_parwise(self):
        if self.A.shape != self.B.shape:
            print("Error: Pairwise failed!")

    def euclidean_dist_squared(self):
        """Computes the Euclidean distance between rows of 'A' and rows of 'B'

        Parameters
        ----------
        A : an N by D numpy array or 1 D array
        B: an T by D numpy array or 1 D array

        Returns: an array of size N by T containing the pairwise squared Euclidean distances.
        """
        self.check_vec2matrix()

        return np.sum(self.A**2, axis=1)[:,None] + np.sum(self.B**2, axis=1)[None] - 2 * np.dot(self.A,self.B.T)

    def euclidean_dist_squared_axis(self,axis=0):
        """Computes the Euclidean distance between 'A' and 'B' along spec axis


        Parameters
        ----------
        A : an N by D numpy array or 1D array
        B: an N by D numpy array or 1D array
        axis: 0 along the row, 1 along the column, default is along the column

        Returns: an array of size N by 1(axis=1) or 1 by D(axis=0) containing the pairwise along spec axis squared Euclidean distances.
        """
        self.check_parwise()

        return np.sum((self.A-self.B)**2,axis=axis)

    def dist_squared_sum(self):
        self.check_parwise()
        return np.sum((self.A-self.B)**2)

```

```python
    def cosine_dist(self,X1,X2):

        norm1 = np.sum(X1**2,axis=1)[:,None]
        norm2 = np.sum(X2**2,axis=1)[None]
        norm1[norm1 == 0.0] = 1.0
        norm2[norm2 == 0.0] = 1.0
        dist2 = np.dot(X1,X2.T)**2/norm1/norm2

        dist2 = 1 - dist2
        check_array(dist2)
        return dist2


#Return the length of row with non-null elements
def len_notnull(I):
    return I[0][pd.notnull(I[0])].shape[0]

#Return dynamics data: dr, dtan of the car: num=num (default is agent:0)
def dynamics_data(I,t,num=0):
    ind_x = 2*num
    ind_y = 2*num+1
    if t >0:
        dx = I[t,ind_x]-I[t-1,ind_x]
        dy = I[t,ind_y]-I[t-1,ind_y]
        dr = np.sqrt(dx**2 + dy**2)
        if t >1:
            dx0 = I[t-1,ind_x]-I[t-2,ind_x]
            dy0 = I[t-1,ind_y]-I[t-2,ind_y]

            if dx ==0.0:
                tan = 999
            else:
                tan = dy/dx
            if dx0 ==0.0:
                tan0 =999
            else:
                tan0 = dy0/dx0
            dtan = tan - tan0
        else:
            dtan =0.0
    else:
        dr = 0.0
        dtan =0.0
    return dr,dtan

#return dynamics data over all timesteps
#if truncate = True, all data are set starting from timestep=2 (time=-800)
def dynamics_time_data(I,ind_tar,truncate=True,min_t=0,max_t=9):
    ind_x = 2*ind_tar
    xy_t = I[:,ind_x:ind_x+2]
    dxy_t = xy_t[1:]- xy_t[:-1]
    ddxy_t = dxy_t[1:]- dxy_t[:-1]
    r_t = - np.linalg.norm(xy_t,axis=1)
    dr_t = r_t[1:]- r_t[:-1]
    ddr_t =  dr_t[1:]- dr_t[:-1]

    if truncate:
        res_xy_t,res_dxy_t,res_ddxy_t,res_r_t,res_dr_t,res_ddr_t = xy_t[2:], dxy_t[1:], ddxy_t, r_t[2:], dr_t[1:], ddr_t

        res_xy_t,res_dxy_t,res_ddxy_t,res_r_t,res_dr_t,res_ddr_t = res_xy_t[min_t:max_t],res_dxy_t[min_t:max_t],res_ddxy_t[min_t:max_t],res_r_t[min_t:max_t],res_dr_t[min_t:max_t],res_ddr_t[min_t:max_t]

    return res_xy_t,res_dxy_t,res_ddxy_t,res_r_t,res_dr_t,res_ddr_t
```

```python

def dynamics_time_data_t0(I,ind_tar,truncate=True,t=0):
    ind_x = 2*ind_tar
    xy_t = I[:,ind_x:ind_x+2]
    dxy_t = xy_t[1:]- xy_t[:-1]
    ddxy_t = dxy_t[1:]- dxy_t[:-1]
    r_t = - np.linalg.norm(xy_t,axis=1)
    dr_t = r_t[1:]- r_t[:-1]
    #dr_t = np.linalg.norm(dxy_t,axis=1)
    ddr_t =  dr_t[1:]- dr_t[:-1]
    # ddr_t = np.linalg.norm(ddxy_t,axis=1)
    if truncate:
        return xy_t[2:][t], dxy_t[1:][t], ddxy_t[t], r_t[2:][t], dr_t[1:][t], ddr_t[t]

def dynamics_tavg_data(I,ind_tar):
    xy_t,dxy_t,ddxy_t,r_t,dr_t,ddr_t =dynamics_time_data(I,ind_tar,truncate=True,min_t=0,
    max_t=9)
    return xy_t.mean(axis=0),dxy_t.mean(axis=0),ddxy_t.mean(axis=0),r_t.mean(),dr_t.mean(),
    ddr_t.mean()


def range_t(a):
    return np.arange(-100*(len(a)-1),100,100)

# def range_plot_tx(x):
#     le=len(x)
#     tx = np.zeros((le,2))
#     tx[:,0] =np.arange(-100*(le-1),100,100)
#     tx[:,1] =x
#     return tx.T







def Coor_T_xy_array(xy0,xy1,dxy0,dxy1):
    a = dxy0[:,1]/dxy0[:,0]
    c = dxy1[:,0]/dxy1[:,1]
    a = normalize_slope(a)
    c = normalize_slope(c)
    b = xy0[:,1] - a*xy0[:,0]
    d = xy1[:,0] - c*xy1[:,1]
    x_T = (c*b+d)/(1-a*c)
    x_T = normalize_slope(x_T)
    y_T = (a*d+b)/(1-a*c)
    y_T = normalize_slope(y_T)
    return x_T,y_T


def Coor_T_xy(xy0,xy1,dxy0,dxy1):
    a = dxy0[1]/dxy0[0]
    c = dxy1[0]/dxy1[1]
    a = normalize_slope_sc(a)
    c = normalize_slope_sc(c)
    b = xy0[1] - a*xy0[0]
    d = xy1[0] - c*xy1[1]
    x_T = (c*b+d)/(1-a*c)
    x_T = normalize_slope_sc(x_T)
    y_T = (a*d+b)/(1-a*c)
    y_T = normalize_slope_sc(y_T)
    return x_T,y_T

def normalize_slope(array):
```

```python
    array[np.isnan(array)]=999
    return array

def normalize_slope_sc(scaler):

    if np.isnan(scaler):
        return 999
    else:
        return scaler



def I_inds(I,inds,ind_age=0,include_age=False):
    ind_x=inds*2
    ind_y=inds*2+1
    ind_xy=np.concatenate((ind_x[:,None],ind_y[:,None]),axis=1).flatten()
    ind = ind_xy
    if include_age:
        ind_xy_age=np.array([ind_age*2,ind_age*2+1])
        ind = np.concatenate((ind_xy_age,ind_xy))
    return I[:,ind]

def get_rmse(A,B,type_obj='traj'):
    if A.ndim !=2:
        print("Error: input is not matrix.")
    if A.shape != B.shape:
        print("Error: shape not consistent: A(%s),B(%s)."%(A.shape,B.shape))
    if type_obj=='traj':
        n,d=A.shape
        rmse =np.sqrt(np.sum((A-B)**2)/n/d)
    return rmse


def sign_able(x):
    y=np.sign(x)
    return -(y-1)/2
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Dec  6 15:12:07 2020

@author: xinyao fan
"""

import numpy as np
import pandas as pd
import cmath
from auto_regression import AutoReg
from read_data import read_data
#from raw data to the numerical matrix 11*20 each row is a time slice.
#set all non-exist cars coordinates to nan and shift to the right most col.
def data_prepare_X(X_raw):
    values=X_raw.values
    ind_age=int(np.where(values[0]==' agent')[0]) #index of agent
    age_xy=values[:,(ind_age+2):(ind_age+4)]

    #get non-exist car indices (shift to role indices)
    ind_nan=np.unique(np.where(values[:,6::6]==0)[1])
    ind_nan=(ind_nan+1)*6-4

    #index, data of others, remove all non-exist indices
    ind_oth=np.asarray(np.where(values[0]==' others')[0])
    ind_oth=np.delete(ind_oth,np.where(ind_oth==ind_nan))

```

```python
    ind_oth_xy=np.concatenate((ind_oth+2,ind_oth+3)).T.flatten()
    oth_xy = values[:,ind_oth_xy]

    #11*20 array, the first two columns are coordinates of agent car
    #while the left colums are coordiates of others car
    #fill rest of the dataset with NaN
    nan_xy=np.empty((len(X_raw),2*len(ind_nan)))
    nan_xy[:]=np.NaN
    X=np.concatenate((age_xy, oth_xy, nan_xy), axis=1)
    return X.astype(float)

def data_prepare_y(y_raw):
    le = len(y_raw)
    values=y_raw.values
    y_x = values[:,1]
    y_y = values[:,2]
    if le!= 30:

        AR_model_x = AutoReg(lags=2)
        AR_model_x.fit(y_x)
        y_x_pred=AR_model_x.predict(le,29)

        AR_model_y = AutoReg(lags=2)
        AR_model_y.fit(y_y)
        y_y_pred=AR_model_y.predict(le,29)

        y_x = np.concatenate((y_x,y_x_pred))
        y_y = np.concatenate((y_y,y_y_pred))
    if len(y_x)!=30:
        print("Error: not 30!")
    y=np.zeros((30,2))
    y[:,0] = y_x
    y[:,1] = y_y
    return y


def data_preprocess(dataset,type_dataset='X'):
    dataset_processed=[]

    for i in range(len(dataset)):
        if type_dataset=='X':
            dataset_processed.append(data_prepare_X(dataset[i]))
        elif type_dataset=='y':
            dataset_processed.append(data_prepare_y(dataset[i]))

    return dataset_processed

def data_read_preprocessed():
    X_train,y_train,X_val,y_val,X_test=read_data()
    X_train = data_preprocess(X_train)
    y_train = data_preprocess(y_train,type_dataset='y')
    X_val = data_preprocess(X_val)
    y_val = data_preprocess(y_val,type_dataset='y')
    X_test = data_preprocess(X_test)
    return X_train,y_train,X_val,y_val,X_test


#(x,y) to (rho,theta) vector form
def polar_transform(x):
    #x=f1[:,0:2]
    polar=[]
    for i in range(x.shape[0]):
        tem=complex(x[i,0],x[i,1])
        cn=cmath.polar(tem)
        polar=np.append(polar,cn)
```

```
    polar=np.reshape(polar,(x.shape[0],x.shape[1]))
    return polar

#absolute coordinate system: the intersection is origin.
def polar_system(f): #numerical matrix
    result=np.zeros((f.shape[0],f.shape[1]))
    num_cols = f.shape[1]
    for i in np.arange(0,num_cols,2):
            print(f[:,i:(i+2)])
            anded = polar_transform(f[:,i:(i+2)])
            result[:,i:(i+2)]=anded
    return result

#relative coordinate system: the agent car is the origin
def relative_polar_system(f): #numerical matrix
    result=np.zeros((f.shape[0],f.shape[1]))
    num_cols = f.shape[1]
    for i in np.arange(2,num_cols,2):
        tem=f[:,i:(i+2)]-f[:,0:2]
        anded = polar_transform(tem)
        result[:,i:(i+2)]=anded
    return result
```