CS5691 Pattern Recognition and Machine Learning

Data Contest Nov-2020 Report

Authors

Abhijeet Ingle (AE17B016)

Avinash G Bagali (AE17B110)

Instructor

Dr. Harish Guruprasad Ramaswamy

1 Introduction

The main aim for the PRML (Pattern Recognition and Machine Learning - CS5691) Nov-2020 data contest was to build prediction models for members of Biker-Interest-Group and try to predict which biker-tours will be of interest to bikers. The metadata about the biker's previous interests, his/her demographic details, past tours, friend circle etc was given as well to be used for the model building.

The overall approach decided, after cleaning and visualizing the data-sets (both train and the metadata of tours and bikers), was to impute the data points in the train data-set using the meta data and then treat the problem as a supervised classification task. The reasoning behind the imputation was to try to minimize the heavy class imbalance that was present in the training data with like or 1 (as it was chosen to represent) had 87.69 % of the total points in the cleaned train data-set (label for this set was available).

The following steps cover the entire process of predictive model building and testing:

1. Determining the question to be answered by the model in Machine learning terms
2. Data Cleaning and Visualization
3. Imputation of the missing data in the training data-set
4. Model selection and Hyper-parameter tuning
5. Submission file creation

In this report we explain the procedure adopted in a detailed manner.

2 Approach

2.1 Stating the problem in Machine Learning terms

The problem statement of the contest suggested on classifying the tours a given biker group would like or dislike. As the metadata for the bikers, such as their previous interests, demographic details, past tours and network along with supervised train data was also given it made sense to treat the problem as a supervised classification problem.

The metric used to evaluate this classifier is given as the Mean Average Precision (MAP@K). The definition of this metric as provided in the contest itself is as follows:

$$AP@K(biker\_id) = \frac{1}{GTP} \sum_{k=1}^{min(K,m)} rel(k).P(k)$$

where GTP is the total number of ground truth positives, m is the number of tour_id corresponding to the biker, $P(k)$ is the precision at the cutoff k, $rel(k)$ is a relevance function. The relevance function is just an indicator function which equals 1 if the tour_id at rank k is relevant and equals to 0 otherwise. For illustration we select K as for evaluation, Mean Average Precision is the calculated mean of AP@K over all bikers.

$$MAP@K = \frac{1}{N} \sum_{i=1}^{N} AP@K$$

From the metric we see that out of all the tour_id corresponding to the given biker_id if the classifier can correctly predict the tours the biker will like, and place it at a higher priority we get a high score.

2.2 Data Cleaning and Visualization

The given data-sets are:

1. train.csv
   It has rows corresponding to tours shown to a biker, and data about whether he/she liked the tour or not.

2. test.csv
   Same column entries as the training data-set except the like/dislike columns.

3. tour_convoy.csv
   It consists the list of bikers that showed interest in a particular tour.

4. bikers.csv
   It contains feature information about the bikers.

5. tours.csv
   It contains feature information about the tours.

6. bikers_network.csv
   bikers_network consists of the social networks of the bikers. This derived from the group of bikers that know each other via some groups.

2.2.1 Data description and visualization

- train.csv
  The training data-set given consists of 13866 entries with no-null entries in the first four columns. However there are missing entries in the last two like and dislike columns. Specifically about 10023 data points have no like or dislike label, which is around 72.284 %. Also in the remaining data points for which label is given, class 'like' or 1 (as was chosen to represent it) makes up for 87.69 % of the binary classes. The python code for the above is:

```
import pandas as pd
import numpy as np

def give_info(df):
'''
Gives information about the pandas dataframe 'df'
'''
    print('head')
    print(df.head())
    print()
    print('info')
    print(df.info())
    print()
    print('describe')
    print(df.describe())
    return None

train = pd.read_csv('.../data/train.csv')
give_info(train)
print('numbers of points with no \
label:\t{}'.format(((train.like==0)&(train.dislike==0)).sum()))
print('percent of total data \
points:\t{}'.format(((train.like==0)&(train.dislike==0)).sum()/(train.shape[0])))

#Points for which label is given i.e. either like=1 or dislike=1
train_2 = train[(train.like==1)|(train.dislike==1)].reset_index(drop=True)
print('class 1 percentage:\t{}'.format((train_2.like==1).sum()/train_2.shape[0])
```

- test.csv
  Same function as defined above was used to check the information in the test data-set. The data-set has same initial four columns as the train data-set with 2690 entries.

- tour_convoy.csv
  This data-set consisting of lists of bikers that showed interest in a particular tour. It has 24144 entries with 5 columns namely 'tour_id', 'going', 'maybe', 'invited' and 'not_going'. The missingno matrix plot along with code is given below:

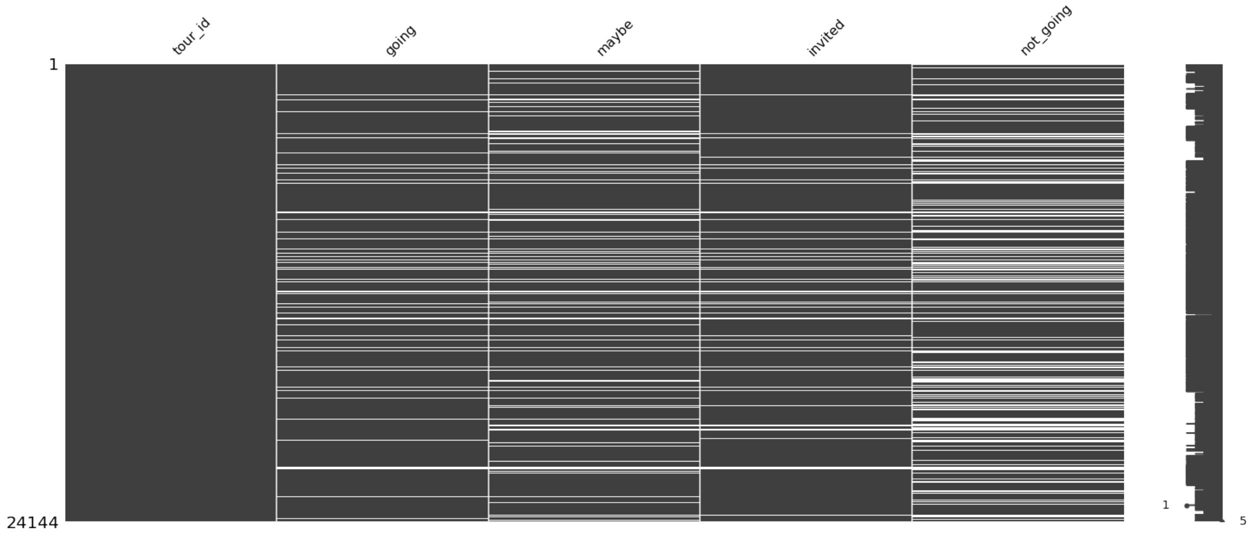


Figure 1: tour convoy missing values visualization

```
import missingno as msg
import matplotlib
%matplotlib inline

tour_convoy = pd.read_csv('../data/tour_convoy.csv')
give_info(tour_convoy)
print('NaN entries in data-set\n{}'.format(tour_convoy.isnull().sum()))
msg.matrix(tour_convoy)
```

  As can be seen in the fig 1, all the columns except the 'tour_id' one as missing entries. To clean it up, we simply dropped the rows with NaN values, bringing the total count of rows from 24144 entries to 16787 entries.

- bikers.csv
  The dataset containing feature information about the bikers was checked and visualized using the missingno library and the matplotlib backend. This is shown along with the code as follows:

```
bikers_ = pd.read_csv('.../data/bikers.csv')
give_info(bikers_)
```

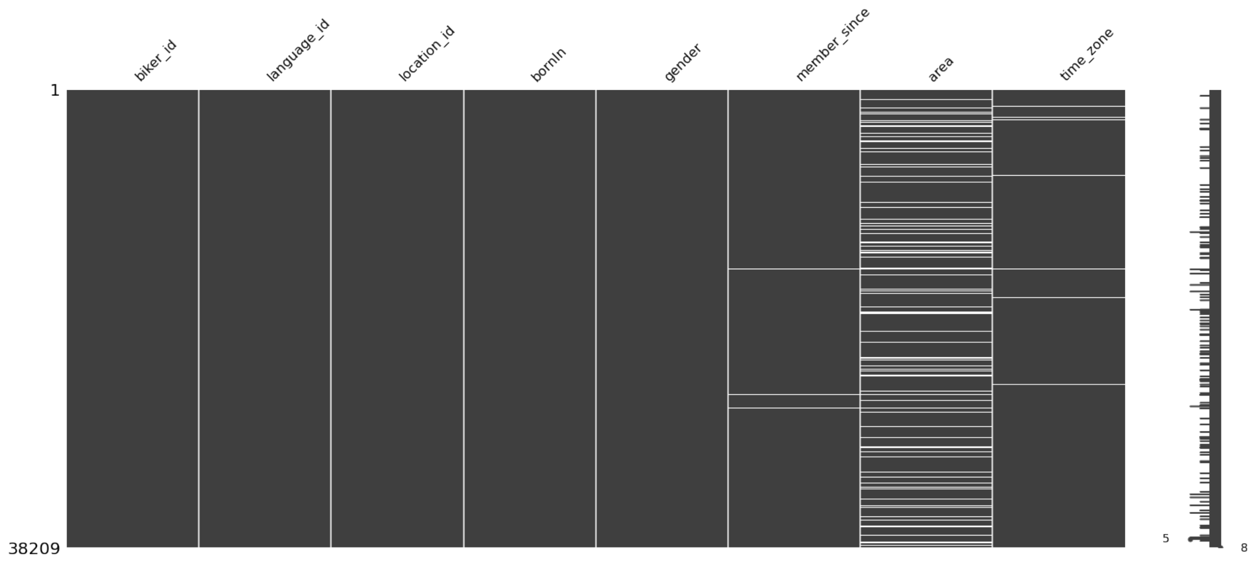


Figure 2: bikers dataset missing values

```
msg.matrix(bikers_)
print('null values in the data-set\n{}'.format(bikers_[['members_since', 'area', \
'time_zone']].isnull().sum()))
bikers_.drop(columns='area',inplace=True)
bikers_.dropna(inplace=True)
bikers_.reset_index(drop=True)
```

This data-set has 38209 entries or rows with 8 columns as shown in the fig 2. The 'member_since', 'area' and the 'time_zone' columns are the ones that needs attention as they have missing entries in them, with 57, 5464 and 436 missing entries respectively. We dropped the 'area' column from the bikers data-set also dropped the rows having null entries in 'members_since' and 'time_zone' columns. The final bikers dataset consisted of 37624 entries.

- tours.csv
  This data-set consists feature information about the tours. The data-set is huge with 3,137,972 entries and 110 columns. Out of 110 columns only 6 consist of missing entries, they are visualized in fig 3 and the entries are worked out using similar code as given already.

  The six columns namely 'city', 'state', 'pincode', 'country', 'latitude' and 'longitude' have 42% to 86% of the columns as missing entries. So while cleaning, these columns were dropped to finally get 104 columns and the same 3137972 entries in rows.

- bikers_network.csv
  It consists of the social networks of the bikers. This is derived from the group of bikers that are know each other via some groups. This data-set has 38202 entries and 2 columns.

  This data-set has only 139 missing entries in the 'friends' columns. Suitable cleaning of this data-set was done with dropping the corresponding rows. Also for dealing with qualitative data and time columns we used the scikit-learn library's Ordinal Encoder and datetime module. The specific functions are:

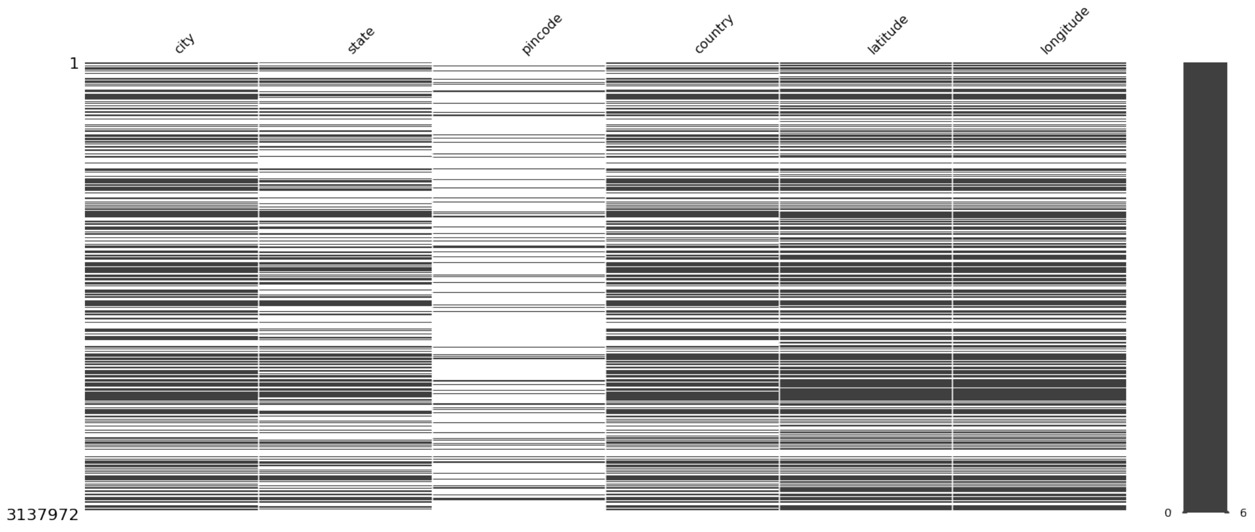


Figure 3: tours missing entries visualization

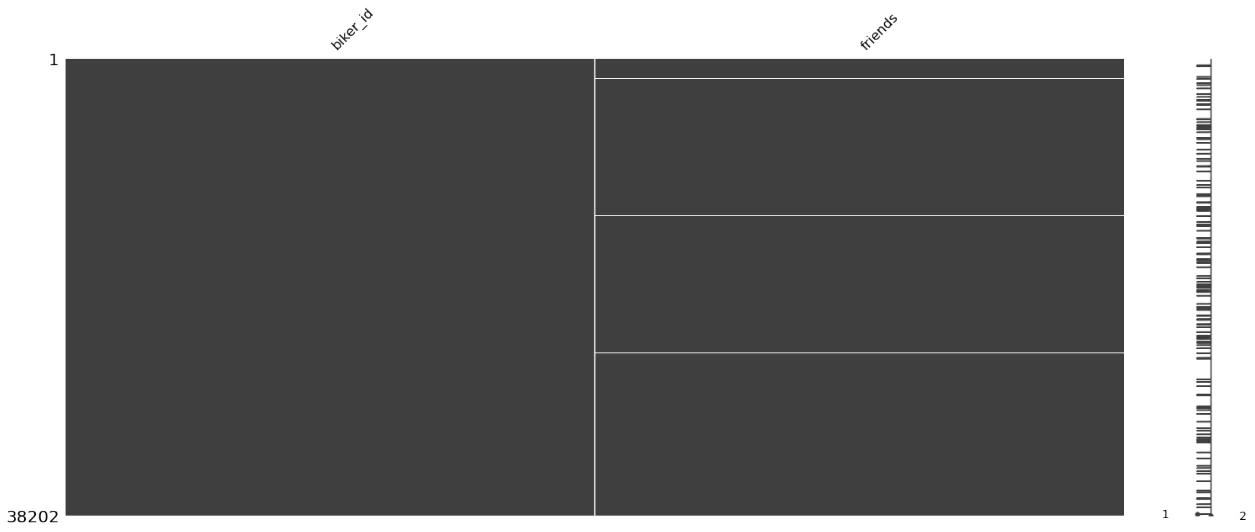


Figure 4: bikers network missing entries visualization

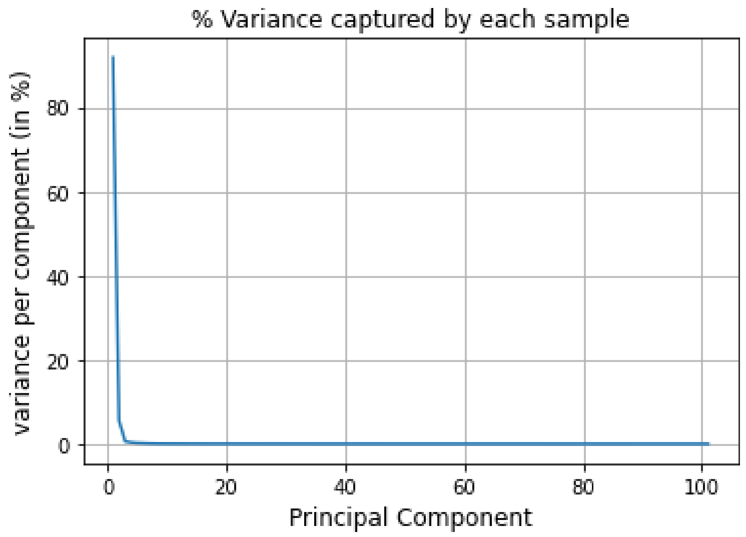
```
import datetime
from sklearn.preprocessing import OrdinalEncoder

def conv_to_time(string):
    my_string = string
    b = datetime.datetime.strptime(my_string, '%d-%m-%Y %H:%M:%S')
    c = datetime.datetime.timestamp(b)
    return c
```

2.3 Imputation of the missing data

2.3.1 PCA on the tours dataset

We notice that the size of the tours dataset is very large due to the 101 word features. this makes the time to compute similar tours very long as it has a 101 features to compare. We observe that the w columns are extremely sparse and thus it might take only a few features to represent the data after performing PCA.

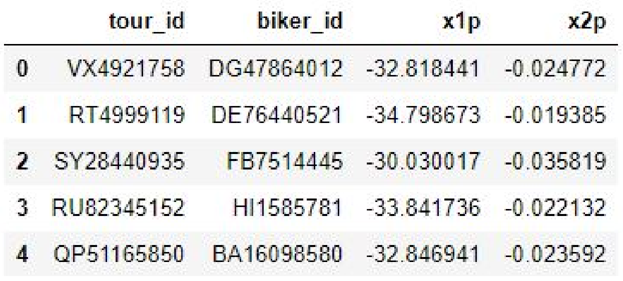
Figure 5:



In the above figure, we the fraction of the total variance that is captured by each of the 101 Principal components. After further analysis , we decide that 2 Principal components is the perfect trade-off between variance captured and the computational cost ,as it caputers 97.34% of the total variance.

We replace the 101 w columns with the 2 PCA components named as 'x1p' and 'x2p' in the figure given below

Figure 6: x1p and x2p are first 2 Principal components



	tour_id	biker_id	x1p	x2p
0	VX4921758	DG47864012	-32.818441	-0.024772
1	RT4999119	DE76440521	-34.798673	-0.019385
2	SY28440935	FB7514445	-30.030017	-0.035819
3	RU82345152	HI1585781	-33.841736	-0.022132
4	QP51165850	BA16098580	-32.846941	-0.023592

We also decide to drop the following columns from the the data 'city','state', pincode','country' as they all encode the location of the tour, which can be much accurately given by the latitude and the longitude. However, we recognize that very few tours have the latitude and longitude data. We impute this information using the location data for the organizing biker (biker_id corresponding to the particular tour) in locs.csv that is the latitude and longitude data for each biker using the geopy library. Once we have done this we can cluster the tours together to get the most similar tours for each tour.

2.3.2 Imputing like/dislike data in train data

We will now impute the missing like or dislike data in the train data using the bikers network and convoy. We create a a common column called 'like/dislike' which will be +1 to denote a like and -1 to denote a dislike and encode the missing data as 0 temporarily. So our job is to impute the 0 zero values in the train data to either +1 or -1 using the bikers network and tours convoy data.

We impute the train data by using a score system. We iterate through the train_missing data (rows of train such that 'like/dislike' == 0). To find the score for a biker and tour pair, we pass the value of the current biker and current tour to the find_score function.

We divide the scoring into a primary and secondary score. If the primary score gives us a decisive result, we predict using just the primary result. However, if the score is still zero after the primary scoring, we turn to the secondary scoring to help us predict. The reason for this split is the secondary score isn't as intuitive as the primary one and also has a higher time complexity. Thus we can save some run time by making this split.

In the code snippet below, we see the primary scoring method. for a given biker and tour pair. We store all the biker's friends obtained from the network into temp1. temp2 and temp3 are the lists of the bikers not going and going to the particular tour respectively, which is obtained from the tours convoy data. invitees is the list of bikers invited to the given tour. Now we iterate thorough temp1(biker's friends) and increase the score if the biker's friend is going to the tour or is invited to the tour and decrease the score if the biker's friend is not going to the tour. After iterating thorough all the biker's friends, if score is non-zero we return the sign of the score. +1 meaning like and -1 meaning dislike. If we get a zero primary score, we go to calculating the secondary score.

```
def find_score(biker,tour):
    org = tours_mod['biker_id'][tour]           #organizer of the given tour


    #temp1 contains the list of friends of the biker
    if type(network['friends'][biker]) == str:
        temp1 = network['friends'][biker].split(" ")
    else:
        temp1 = []



    #temp2 contains a list of names of bikers not going to the given tour
    if type(convoy['not_going'][tour]) == str:
        temp2 = convoy['not_going'][tour].split(" ")
    else:
        temp2 = []

    #temp3 contains a list of names of bikers going to the given tour
    if type(convoy['going'][tour]) == str:
        temp3 = convoy['going'][tour].split(" ")
    else:
        temp3 = []

    #invitees contains a list of names of bikers invited to the given tour
    inv_flag = False
    if convoy['invited'][tour] == 'str':
        invitees = invitees.split(" ")
        inv_flag = True


    score = 0
    for i in range(len(temp1)):             #iterating over the list of friends of the curr biker

        if temp1[i] in temp3:               #friends of given biker going to the given tour
            score += 2
        if temp1[i] in temp2:               #friends of given biker not going to the given tour
            score -= 2
        if inv_flag and temp[i] in invitees:  #friends of given biker invited to the given tour
            score += 1
```

If the primary score is zero, we turn to calculating a secondary score which takes into account the following:

- If the organizer of the given tour is the friend of the biker
- If the tour is being organized in the same area as that of the biker.
- Using the rating history of the biker for other tours. (we'll discuss in detail below)

```
    #if the score is either +ve or -ve we return the sign.
    if score != 0:
        return np.sign(score)
    else:
        if org in temp1:                              #if organizer is biker's friend
            return 1

        bik_area = bikers['area'][biker]     #if tour is in the same area as the biker
        if org in bikers.index:
            tou_area = bikers['area'][org]
            if bik_area == tou_area:       #if the tour is in the bikers area the biker
                return 1                   #is more likely to go
```

The secondary score uses the rating history of the given biker to predict whether or not, he'll like this tour. other_rats contains the rating history of the biker derived from the train data. We now iterate through these tours and generate a secondary score. similar to what we did before temp2_ and temp are the list of bikers not going and going to the tour rated by our biker respectively.Now we iterate through temp1 (friends of the biker). and in a similar fashion to before we increase the score if a biker's friends is going to the tour and decrease the score if the biker's friend is not going to the rated tour. We can use the tours that are similar to the tour using the clustering done by using the PCA reduced ws and the latitude and longitude of the biker. If the secondary score is non-zero,then we return the sign of the score else we return -1. If we fail to find any other similar tour to the given tour, then we simply return -1.

```
    #other tours the biker has rated
    other_rats = train_present[train_present['biker_id'] == biker]
    if len(other_rats != 0):
        score_ = 0        #secondary score based on the other ratings of the biker to other tours
        for j in range(len(other_rats)):

            #temp2_ is the list of bikers not going to one of the tours that our curr
            #biker has rated before

            if type(convoy['not_going'][other_rats['tour_id'][other_rats.index[j]]]) == str:
                temp2_ = convoy['not_going'][other_rats['tour_id'][other_rats.index[j]]].split(" ")
            else:
                temp2_ = []

            #temp2_ is the list of bikers going to one of the tours that our curr biker
            #has rated before

            if type(convoy['going'][other_rats['tour_id'][other_rats.index[j]]]) == str:
                temp3_ = convoy['going'][other_rats['tour_id'][other_rats.index[j]]].split(" ")
            else:
                temp3_ = []

            for i in range(len(temp1)):
                if temp1[i] in temp3_:#if bikers friend is going to the tour rated by our biker
                    score_ += 2
                if temp1[i] in temp2_:#if bikers friend is not going to the tour rated by our biker
                    score_ -= 2

        #if non zero score ,return sign of the score
        if score_ != 0:
            return np.sign(score_)
        else:
            return -1                           #if still nothing, the biker mostly wouldn't like
    else:
        return -1                               #if our biker hasn't rated any other tour , return dislike
```

The final train data-set was constructed by the concatenating the original train data-set with no missing entries and only class 1 present with the imputed train data-set which we got earlier with the other class 0. i.e.

```
train_original = train[[(train.like==1)]]
train_original['like/dislike] = train_original['like']
train_original.drop(columns=['like', 'dislike'],inplace=True)
train_final = pd.concat([train_original, \
train_imputed.loc[train_imputed['like/dislike']==-1,:]], axis=0)
```

2.4 Model selection and Hyper-parameter tuning

The classifiers tested with the final train data-set for classifying into like (1) and dislike (-1) classes were K-nearest neighbors, AdaBoost classifier and the Support Vector Classifier. The tuning and testing is done using the 5-fold cross validation metric in the GridSearchCV class of sklearn library.The code is as follows:

```
from sklearn.processing import OrdinalEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier as KNC
from sklearn.svm impot SVC

models = [KNC(), AdaBoostClassifier(), SVC()]
param_grid_list =  [{'n_neighbors' : range(5,101,5),'weights':('uniform', 'distance')}, \
{'base_estimator' : [LogisticRegression(), \
DecisionTreeClassifier(max_depth=1)],'n_estimators':[50, 100, 150]}, {'C':[0.1, 1, 10],\
'kernel':['linear', 'rbf', 'poly'], 'class_weight': ['balanced']}]

best_estimator_out = models[0]
best_score_out = -1e-10

#preparaing xtrain and ytrain
x_train = train_final.drop(columns=['like/dislike', 'like',\
'dislike']).reset_index(drop=True)
y_train = train_final['like/dislike'].copy().reset_index(drop=True)
```

```
x_train.timestamp = x_train.timestamp.apply(conv_to_time)
x_train.timestamp = (x_train.timestamp - x_train.timestamp.mean())/x_train.timestamp.std()

#encoding the qualitative data
enc = OrdinalEncoder()
x_enc_qual = pd.DataFrame(enc.fit_transform(x_t[['biker_id', 'tour_id']]),\
columns=['biker_id', 'tour_id'])
x_enc = x_train
x_enc.tour_id = x_enc_qual.tour_id
x_enc.biker_id = x_enc_qual.biker_id

#change the dislike label from -1 to 0
y_train[y_train ==-1] = 0

for m, p in zip(models, param_grid_list):
    grid = GridSearchCV(m, p)
    grid.fit(x_enc, y_train)
    if grid.best_score_ > best_score_out:
        best_score_out = grid.best_score_
        best_estimator_out = grid.best_estimator_

print('best estimator:\t{}'.format(best_estimator_out))
print()
print('best score:\t{}'.format(best_score_out))
```

The best classfier we get is AdaBoost Classifier with base estimator as the Logistic Regression algorithm. The score is 0.5215 as can be checked from the above code

2.5 Submission file creation

The below code was used to produce the submission file to be submitted to the contest

```
class submission:
    def __init__(self, xtrain, ytrain, xtest):
        self.train_x = xtrain
        self.train_y = ytrain
        self.test_x = xtest

    def create_submission_1(self,clf):
        classifier = clf
        enc = OrdinalEncoder()
        x_train_enc_cols = pd.DataFrame(enc.fit_transform(self.train_x[['biker_id', 'tour_id']]), colu
        x_train_enc = self.train_x.copy()
        x_train_enc['biker_id'] = x_train_enc_cols['biker_id']
        x_train_enc['tour_id'] = x_train_enc_cols['tour_id']
        classifier.fit(x_train_enc, self.train_y)
        x_enc_cols = pd.DataFrame(enc.fit_transform(self.test_x[['biker_id', 'tour_id']]), columns=['b
        x_enc = self.test_x.copy()
        x_enc['biker_id'] = x_enc_cols['biker_id']
        x_enc['tour_id'] = x_enc_cols['tour_id']
        x = self.test_x.copy()
        ypred = pd.Series(classifier.predict(x_enc),name='ypred')
        ans = pd.concat([x,ypred],axis=1)
        sub = []
        for biker in x.biker_id.drop_duplicates():
            like_tours = ans.tour_id[np.logical_and(ans.biker_id==biker, ans.ypred==1)].to_numpy()
            dislike_tours = ans.tour_id[np.logical_and(ans.biker_id==biker, ans.ypred==0)].to_numpy()
            temp = [biker, " ".join(like_tours.tolist()+dislike_tours.tolist())]
            sub.append(temp)
        final_sub = pd.DataFrame(data=sub,columns=['biker_id','tour_id'])
        return final_sub

    def create_submission_2(self,clf):
        classifier = clf
        enc = OrdinalEncoder()
        x_train_enc_cols = pd.DataFrame(enc.fit_transform(self.train_x[['biker_id', 'tour_id']]), colu
        x_train_enc = self.train_x.copy()
        x_train_enc['biker_id'] = x_train_enc_cols['biker_id']
        x_train_enc['tour_id'] = x_train_enc_cols['tour_id']
        classifier.fit(x_train_enc, self.train_y)
        x_enc_cols = pd.DataFrame(enc.fit_transform(self.test_x[['biker_id', 'tour_id']]), columns=['b
        x_enc = self.test_x.copy()
        x_enc['biker_id'] = x_enc_cols['biker_id']
        x_enc['tour_id'] = x_enc_cols['tour_id']
        x = self.test_x.copy()
        ypred = pd.Series(classifier.predict(x_enc),name='ypred')
        ans = pd.concat([x,ypred],axis=1)
        sub = []
        np.random.seed(1)
        for biker in x.biker_id.drop_duplicates():
            like_tours = ans.tour_id[np.logical_and(ans.biker_id==biker, ans.ypred==1)].to_numpy()
            np.random.shuffle(like_tours)
            dislike_tours = ans.tour_id[np.logical_and(ans.biker_id==biker, ans.ypred==0)].to_numpy()
            np.random.shuffle(dislike_tours)
            temp = [biker, " ".join(like_tours.tolist()+dislike_tours.tolist())]
            sub.append(temp)
        final_sub = pd.DataFrame(data=sub,columns=['biker_id','tour_id'])
        return final_sub

clf = AdaBoostClassifier(base_estimator=LogisticRegression())
xtrain = x_t
ytrain = y_t
test.timestamp = test.timestamp.apply(conv_to_time)
test.timestamp = (test.timestamp - test.timestamp.mean())/test.timestamp.std()
sub_ = submission(xtrain, ytrain, test)
sub1 = sub_.create_submission_1(clf)
sub2 = sub_.create_submission_2(clf)
sub1.to_csv('AE17B016_AE17B110_1.csv', header=True, index=False)
sub2.to_csv('AE17B016_AE17B110_2.csv', header=True, index=False)
```

The difference between the two create submission functions is the random shuffling of all the tours classified as 'like' for the biker by the classifier in the second function.We do the shuffling for the 'dislike' tours as well but not on the entire tours corresponding to a biker. If the classification was perfect both the functions would actually create the same file. But as it isn't we get different submission file and consequently scores for both the files.

3 Results and Conclusion

- We compress the tours dataset by applying PCA and retaining only 2 principal components that capture about 97.3% of the total variance.
- This compressed dataset along with the location data from geopy can be used to cluster the tours which is used later in imputing the train data.
- We impute the missing values in train data using the bikers network to obtain the friends of the bikers and the tour convoy and tour_mod (compressed tours) using a primary and secondary scoring method.
- Using the imputed data, The best classifier we get is Ada Boost Classifier with base estimator as the Logistic Regression algorithm.
- One of the contributing factors to a lower performance can be attributed to the class imbalance in the likes and dislikes.
- Possible things that can be done to improve the performance of the classifier would be to extend the dimensionality of the training data-set by adding more features given in the metadata. Also using other algorithms like CatBoost, XGBoost etc could have improved the submission file score.