Contents

1 task1/README

```
.:
README
src/
USERS.txt

./src:
adaboost.py
BinAgent
binarysearch.py
classifier.py
ex4_tools.py
geo
model.py
models.py
strong.py
strong2.py
testmodel.py
weather.py
wether.json
```

2 task1/USERS.txt

```
1  nir.vaknin, 31151478
2  davidponar, 208504050
3  nitsanwoja, 314967837
4  amirlevy95, 204880330
```

3 task1/src/adaboost.py

```
"""
=================================================
     Introduction to Machine Learning (67577)
=================================================

Skeleton for the AdaBoost classifier.

Author: Gad Zalcberg
Date: February, 2019

"""
import numpy as np
from ex4_tools import *
from matplotlib import pyplot as plt

class AdaBoost(object):

    def __init__(self, WL, T , support_wights = False):
        """
        Parameters
        ----------
        WL : the class of the base weak learner
        T : the number of base learners to learn
        """
        WL, self.fe = WL
        self.WL = [ WL  for _ in range(T)]
        self.T = T
        self.h = [None]*T     # list of base learners
        self.w = np.zeros(T)  # weights
        self.support_wights = support_wights

        # self._predict = np.vactorize(WL.prdeict)

    def train(self, X, y):
        """
        Parameters
        ----------
        X : samples, shape=(num_samples, num_features)
        y : labels, shape=(num_samples)
        Train this classifier over the sample (X,y)
        After finish the training return the weights of the samples in the last iteration.
        """
        y = y.flatten()
        D = np.ones( len( y ) ) * 1/len( y )

        def find_loss_D(h, D, X, y):
            _out = h.predict(X).flatten()
            return np.sum( D[_out != y.transpose()] ), _out

        def normalize(vec):
            r = np.sum(vec )
            return vec / r if r != 0 else 0

        self.w = []

        for i in range(self.T):
            self.h[i] = self.WL[i](self.fe)
            if self.support_wights:
                self.h[i].train(X, y, D)
```

```
            else:
                self.h[i].train(X, y)

            e_t, _out = find_loss_D(self.h[i], D, X, y)
            # print("[%]et:")
            # print(e_t)
            w = 0.5 * np.log( 1/e_t - 1 )
            D = normalize(D * np.e **( -w * y * _out ) )
            self.w.append(w)

        self.w = np.array(self.w)
        return D

    def predict(self, X, max_t=None):
        """
        Parameters
        ----------
        X : samples, shape=(num_samples, num_features)
        :param max_t: integer < self.T: the number of classifiers to use for the classification
        :return: y_hat : a prediction vector for X. shape=(num_samples)
        Predict only with max_t weak learners,
        """
        if max_t is None:
            max_t = self.T
        return np.sign(
                 sum(
                  self.w[i] * self.h[i].predict(X) for i in range(max_t)) )

    def error(self, X, y, max_t=None):
        """
        Parameters
        ----------
        X : samples, shape=(num_samples, num_features)
        y : labels, shape=(num_samples)
        :param max_t: integer < self.T: the number of classifiers to use for the classification
        :return: error : the ratio of the correct predictions when predict only with max_t weak learners (float)
        """
        if max_t is None:
            max_t = self.T
        errors = sum(np.ones( len(y) )[ self.predict(X, max_t) != y ])
        return errors / len(y)


class AdaBoostList(AdaBoost):
    def __init__(self, WLs):
        self.WL = WLs
        self.T = len(WLs)
        self.h = [None]*self.T   # lisself. of base learners
        self.w = np.zeros(self.T)  # weights

    def prdict(self, X):
        return super().prdict(X, self.T)


def test():
    def plot_as_function_of_agents(A, X, y, max_t):
        yplot = [A.error(X, y, t) for t in range(max_t)]
        plt.plot([t for t in range(max_t)], yplot)

    def plot_points(A, X, y,
     classfiers=[ 5, 10, 50, 100, 200, 500 ], D_T=None, noise=0):

        for i ,T in enumerate(classfiers):
            plt.subplot( 320 + i + 1)
            decision_boundaries(A, X, y, num_classifiers=T, weights=D_T)

        size_str = True if D_T is not None else False
        plt.savefig(f"plot_points_{size_str}_{noise}.png")
```

```
128
129     for noise in [0, 0.01, 0.4]:
130         A = AdaBoost( DecisionStump, 500 )
131         X,y = generate_data(5000, noise)
132         A.train(X,y)
133
134
135         plot_as_function_of_agents(A, X, y, 500)
136         X,y = generate_data(200, noise)
137         plot_as_function_of_agents(A, X, y, 500)
138         plt.legend(["training error rate","testing error rate"])
139         plt.xlabel("agents")
140         plt.ylabel("error rate")
141         plt.ylim([0,1])
142         plt.title(f"error rate above distribution with nosie - {noise}")
143         plt.savefig(f"plot_error_{noise}.png")
144
145
146         D_T = A.w.transpose()
147         D_T = 30* D_T/max(D_T)
148         plt.figure()
149
150         for d_T in [None, D_T ]:
151             plot_points(A, X, y, D_T=d_T, noise=noise)
152             plt.figure()
153
154
155
156 if __name__ == "__main__":
157     test()
```

4 task1/src/binarysearch.py

```python
import numpy as np
import re


class binarysearch:

    def __init__ (self, models):
        self.mods = models
        self.iterations = 6


    '''
         for _bool in re > middle ] )
                y = np.array( [ {  False : [0] , True : [1]  }[ _bool ]
    '''
    # def load_classifers(self, classifers_steps):

    #     pass


    def _predict(self, x, treshold):
        return self.mods[treshold].predict(x)

    def predict(self, X, start_range, end_range):
        times_delay = 0
        for j in range(self.iterations):
            middle =(start_range + end_range)/2
            res = np.zeros( shape = start_range.shape)
            for i, time in enumerate(start_range):
                res[i] = self._predict(X.iloc[[i]] , start_range[i])[0]
            res = (res +  np.ones(len(res)))/2
            start_range, end_range = (1 - res) * start_range + res * middle  , (1 - res) * middle +  res * end_range
        return middle

    def dump(self, _file):
        with open(_file, "w") as f :
            _str = ""
            for treshold, _mod in self._mods.items() :
                _str += f"{treshold}:" +_mod.dump( ) + "\n$"

            _file.write(_str)

def binarysearch_read(_file):
    mods = {}
    for banch in re.split('$', _file.read()):
        treshold, _strmod = re.split(':', banch)
        mods[float(treshold)] = AdaBoost_read( )




# testing the idea.
def foo( y, s , e ):
    middle = (s+e)/2
    return  middle * y  ,  middle + middle * y

if __name__ == "__main__":
    s , e = np.array( [ 0, 0 ,0]), np.array( [ 1, 1 ,1])
    y = np.array( [1, 0, 1] )
```

```
60      print( foo(y, s, e) )
61
62
63
64
```

5 task1/src/classifier.py

```python
# from strong import *
from models import abcModel, DecisionTree, Logistic, SVM, DecisionStumpWarper
from adaboost import AdaBoost, AdaBoostList
from itertools import combinations
import numpy as np
import pandas as pd
from binarysearch import binarysearch
from datetime import date
from random import shuffle
from sklearn.model_selection import train_test_split
from geopy.geocoders import Nominatim, ArcGIS
import json
import math
factor_reason = {'CarrierDelay': 0, 'WeatherDelay': 1, 'NASDelay': 2, 'LateAircraftDelay': 3}

#
# def pre_proc_class_old(_dataset, droped_fe, categorical):
#     y_factor = []
#     _dataset = _dataset.dropna()
#     _dataset['month'] = pd.DatetimeIndex(_dataset['FlightDate']).month
#     _dataset['day'] = pd.DatetimeIndex(_dataset['FlightDate']).day
#     _dataset['year'] = pd.DatetimeIndex(_dataset['FlightDate']).year
#
#     for factor in _dataset["DelayFactor"]:
#         y_factor.append(factor_reason[factor])
#     y_factor = np.array(y_factor)
#     for index, row in _dataset.iterrows():
#         # _dataset.loc[index, 'CRSElapsedTime'] = math.floor(row['CRSElapsedTime'] / 10)
#         _dataset.loc[index, 'CRSArrTime'] = math.floor(row['CRSArrTime'] / 100)
#         _dataset.loc[index, 'CRSDepTime'] = math.floor(row['CRSDepTime'] / 100)
#         _dataset.loc[index, 'Distance'] = math.floor(row['Distance'] / 10)
#
#     cat = pd.DataFrame(pd.get_dummies(_dataset[categorical].astype('category')))
#     _dataset = _dataset.drop(droped_fe + categorical, axis=1)
#     _dataset_prepoc = pd.concat([_dataset.reset_index(drop=True), cat.reset_index(drop=True)], axis=1)
#     return _dataset_prepoc, y_factor



def proc_weather( _dataset ):
    weather = json.loads( open("wether.json").read().replace("'", "\"") )
    for index, row in _dataset.iterrows():
        _date = f"{row['day']:02d}-{row['month']:02d}-{row['year']%2000}"
        if _date in weather:
            for key, val in weather[ _date ].items():
                if key != 'day':
                    row[key] = val



def pre_proc_class(_dataset, categorical):


    _dataset['month'] = pd.DatetimeIndex(_dataset['FlightDate']).month
    _dataset['day'] = pd.DatetimeIndex(_dataset['FlightDate']).day
    _dataset['year'] = pd.DatetimeIndex(_dataset['FlightDate']).year

    proc_weather( _dataset )

```

```
    for index, row in _dataset.iterrows():
        # _dataset.loc[index, 'CRSElapsedTime'] = math.floor(row['CRSElapsedTime'] / 10)
        _dataset.loc[index, 'CRSArrTime'] = math.floor(row['CRSArrTime'] / 100)
        _dataset.loc[index, 'CRSDepTime'] = math.floor(row['CRSDepTime'] / 100)
        _dataset.loc[index, 'Distance'] = math.floor(row['Distance'] / 10)


    geolocator = ArcGIS(username="david.ponarovsky", password="mxSrWYYdSq++J7+",
     referer="efMmfQrfh1o_Ag-MEzx5-en9lLs-m_Vu5T_JU7K45vfUhjEccY6W2cilzPnn2r9TO7fpPnAeK_U8EWptaUiXOC7FosOKaovLPAXQRO6PGnXrUY
     timeout=500000)

    def DynmicGeo( df, _keys ):
        import time
        from pprint import pprint

        #Dynmic = {}
        Dynmic = json.loads( open("geo").read().replace("'", "\"") )
        try :
            for _key in _keys:
                df[f'lat{_key}']=0
                df[f'long{_key}']=0
                for x in range(len(df)):
                    if df[_key][x] in Dynmic:
                        location = Dynmic[df[_key][x]]
                        if type(location) == str:
                            df.at[x, f'lat{_key}'] , df.at[x, f'long{_key}'] = eval(location)
                        else:
                            df.at[x, f'lat{_key}'] , df.at[x, f'long{_key}'] = location
                    else:
                        location = geolocator.geocode(df[_key][x])
                        Dynmic[df[_key][x]] = location.latitude,  location.longitude
                        # time.sleep(2)
                    #print(location , location.latitude, location.longitude)
                        df.at[x, f'lat{_key}']= location.latitude
                        df.at[x, f'long{_key}'] = location.longitude
        except:
            print("except")

        pprint(Dynmic)

        for key, val in Dynmic.items():
            Dynmic[key] = str(val)
        with open('geo', 'w') as f:
            json.dump(Dynmic, f)

        return df


    geoKeys = [ "Origin",
            "OriginCityName",
            "OriginState",
            "Dest",
            "DestCityName",
            "DestState"]

    _dataset = DynmicGeo(_dataset, geoKeys)

    if len(categorical) >0 :
        cat = pd.DataFrame(pd.get_dummies(_dataset[categorical].astype('category')))
        _dataset_prepoc = pd.concat([_dataset.reset_index(drop=True), cat.reset_index(drop=True)], axis=1)
    else:
        _dataset_prepoc = _dataset

    return _dataset_prepoc


def pre_proc_delay(dataset, drop_fe_delay, categorical):
    y = []
```

```
    for _bool in dataset["ArrDelay"] > 0:
        y.append({False: [0], True: [1]}[_bool])
    y = np.array(y)
    dataset = dataset.drop(drop_fe_delay + categorical, axis=1)
    return dataset, y


def pre_proc_factor(dataset, drop_fe_factor, categorical):
    y_factor = []
    dataset = dataset.dropna()
    for factor in dataset["DelayFactor"]:
        y_factor.append(factor_reason[factor])
    y_factor = np.array(y_factor)
    dataset = dataset.drop(drop_fe_factor + categorical, axis=1)
    return dataset, y_factor


def pred_trees(x, y, x_test, y_test):
    from sklearn import datasets
    from sklearn.metrics import confusion_matrix
    from sklearn.model_selection import train_test_split

    # dividing X, y into train and test data

    # training a DescisionTreeClassifier
    from sklearn.tree import DecisionTreeClassifier
    for depth in [4, 5, 7, 10, 13, 15, 17, 18, 19, 20]:
        dtree_model = DecisionTreeClassifier(max_depth=depth).fit(x, y)
        dtree_predictions = dtree_model.predict(x_test)
        print(dtree_model.score(x_test, y_test))
        print(depth)


def pred_forest(x, y, x_test, y_test):
    from sklearn.ensemble import RandomForestClassifier

    model = RandomForestClassifier(random_state=20)
    model.fit(x, y)
    print(model.score(x_test, y_test))


def identify_corelated_features(df):
    # corr = df.corr()
    # #print(sns.heatmap(corr))
    # columns = np.full((corr.shape[0],), True, dtype=bool)
    # for i in range(corr.shape[0]):
    #     for j in range(i + 1, corr.shape[0]):
    #         if corr.iloc[i, j] >= 0.9:
    #             if columns[j]:
    #                 columns[j] = False
    # selected_columns = df.columns[columns]
    # df = df[selected_columns]
    return df


def pred(x, y, x_test, y_test):
    from sklearn import datasets
    from sklearn.metrics import confusion_matrix
    from sklearn.model_selection import train_test_split

    # training a KNN classifier
    from sklearn.neighbors import KNeighborsClassifier
    for n in [11, 13, 15, 20, 30, 35, 40, 45]:
        knn = KNeighborsClassifier(n_neighbors=n).fit(x, y)

        # accuracy on X_test
        accuracy = knn.score(x_test, y_test)
        print(accuracy)
```

```
        print(n)

    # creating a confusion matrix
    # knn_predictions = knn.predict(x)
    # cm = confusion_matrix(y_test, knn_predictions)


featuers = ["DayOfWeek",
            "FlightDate",
            "Reporting_Airline",
            "Tail_Number",
            "Flight_Number_Reporting_Airline",
            "Origin",
            "OriginCityName",
            "OriginState",
            "Dest",
            "DestCityName",
            "DestState",
            "CRSDepTime",
            "CRSArrTime",
            "CRSElapsedTime",
            "Distance"]

droped_fe_factor = ['Flight_Number_Reporting_Airline',
                    'Tail_Number',
                    'DelayFactor',
                    "OriginCityName",
                    "OriginState",
                    "DestCityName",
                    "DestState", 'FlightDate', "CRSElapsedTime"]

droped_fe_delay = ['Flight_Number_Reporting_Airline',
                   'Tail_Number',
                   "OriginCityName",
                   "OriginState",
                   "DestCityName",
                   "DestState", 'FlightDate', "CRSElapsedTime", 'ArrDelay', 'DelayFactor', 'Origin', 'Dest' ]

categorical_new = ['Reporting_Airline']

def final_pre_proc(_dataset):
    x = pre_proc_class(_dataset, categorical_new)
    x_delay,y_delay = pre_proc_delay(x,droped_fe_delay,categorical_new)
    x_factor,y_factor = pre_proc_delay(x,droped_fe_factor,categorical_new)

    x_factor = identify_corelated_features(x_factor)
    return x_delay, y_delay, x_factor,y_factor

if __name__ == '__main__':
    _dataset = pd.read_csv("~/data/train_data.csv")

    x = pre_proc_class(_dataset, categorical_new)

    print(x)

    x_delay,y_delay = pre_proc_delay(x,droped_fe_delay,categorical_new)
    x_factor,y_factor = pre_proc_delay(x,droped_fe_factor,categorical_new)

    x_factor = identify_corelated_features(x_factor)
    x_train, x_test, y_train, y_test = train_test_split(x_factor, y_factor)
    # print(x['month'])

    # print(y_factor)
    # pred_trees(x_train, y_train, x_test, y_test)
    from datetime import datetime

    # datetime object containing current date and time
    print(datetime.now())
```

```
264         print(x_test.shape)
265         # pred_forest(x_train, y_train, x_test, y_test)
266         print(datetime.now())
267         # pred(x_train, y_train, x_test, y_test)
268         # print(identify_corelated_features(x, y_del))
```

6 task1/src/ex4 tools.py

```
"""
=================================================
     Introduction to Machine Learning (67577)
=================================================

This module provides some useful tools for Ex4.

Author: Gad Zalcberg
Date: February, 2019

"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from itertools import product
from matplotlib.pyplot import imread
import os
from sklearn.model_selection import train_test_split


def find_threshold(D, X, y, sign, j):
    """
    Finds the best threshold.
    D =  distribution
    S = (X, y) the data
    """
    # sort the data so that x1 <= x2 <= ... <= xm
    sort_idx = np.argsort(X[:, j])
    X, y, D = X[sort_idx], y[sort_idx], D[sort_idx]


    thetas = np.concatenate([[-np.inf], (X[1:, j] + X[:-1, j]) / 2, [np.inf]])
    minimal_theta_loss = np.sum(D[y == sign])  # loss of the smallest possible theta
    losses = np.append(minimal_theta_loss, minimal_theta_loss - np.cumsum(D * (y * sign)))
    min_loss_idx = np.argmin(losses)

    return losses[min_loss_idx], thetas[min_loss_idx]


class DecisionStump(object):
    """
    Decision stump classifier for 2D samples
    """

    def __init__(self):
        self.theta = 0
        self.j = 0
        self.sign = 0


    def fit(self, D, X, y):
        """
        Train the classifier over the sample (X,y) w.r.t. the weights D over X
        Parameters
        ----------
        D : weights over the sample
        X : samples, shape=(num_samples, num_features)
        y : labels, shape=(num_samples)
        """
```

```
60          #print(f"X: {X.shape}")
61          loss_star, theta_star = np.inf, np.inf
62          for sign, j in product([-1, 1], range(X.shape[1])):
63              loss, theta = find_threshold(D, X, y, sign, j)
64              if loss < loss_star:
65                  self.sign, self.theta, self.j = sign, theta, j
66                  loss_star = loss
67
68      def predict(self, X):
69          """
70          Parameters
71          ----------
72          X : shape=(num_samples, num_features)
73          Returns
74          -------
75          y_hat : a prediction vector for X shape=(num_samples)
76          """
77          y_hat = self.sign * ((X[:, self.j] <= self.theta) * 2 - 1)
78          return y_hat
79
80
81  def decision_boundaries(classifier, X, y, num_classifiers=1, weights=None):
82      """
83      Plot the decision boundaries of a binary classfiers over X \subseteq R^2
84
85      Parameters
86      ----------
87      classifier : a binary classifier, implements classifier.predict(X)
88      X : samples, shape=(num_samples, 2)
89      y : labels, shape=(num_samples)
90      title_str : optional title
91      weights : weights for plotting X
92      """
93      cm = ListedColormap(['#AAAAFF', '#FFAAAA'])
94      cm_bright = ListedColormap(['#0000FF', '#FF0000'])
95      h = .003  # step size in the mesh
96      # Plot the decision boundary.
97      x_min, x_max = X[:, 0].min() - .2, X[:, 0].max() + .2
98      y_min, y_max = X[:, 1].min() - .2, X[:, 1].max() + .2
99      xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
100     Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()], num_classifiers)
101     # Put the result into a color plot
102     Z = Z.reshape(xx.shape)
103     plt.pcolormesh(xx, yy, Z, cmap=cm)
104     # Plot also the training points
105     if weights is not None:
106         plt.scatter(X[:, 0], X[:, 1], c=y, s=weights, cmap=cm_bright)
107     else:
108         plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright)
109     plt.xlim(xx.min(), xx.max())
110     plt.ylim(yy.min(), yy.max())
111     plt.xticks([])
112     plt.yticks([])
113     plt.title(f'num classifiers = {num_classifiers}')
114     plt.draw()
115
116
117 def generate_data(num_samples, noise_ratio):
118     '''
119     generate samples X with shape: (num_samples, 2) and labels y with shape (num_samples).
120     num_samples: the number of samples to generate
121     noise_ratio: invert the label for this ratio of the samples
122     '''
123     X = np.random.rand(num_samples, 2) * 2 - 1
124     radius = 0.5 ** 2
125     in_circle = np.sum(X ** 2, axis=1) < radius
126     y = np.ones(num_samples)
127     y[in_circle] = -1
```

```
    y[np.random.choice(num_samples, int(noise_ratio * num_samples))] *= -1

    return X, y
```

7 task1/src/geo

```
{"BNA": "(36.11958985, -86.68308703291842)", "SEA": "(47.4475673, -122.3080158569515)", "JAX": "(45.1683631, 3.6168847)", "L
```

8 task1/src/model.py

```python
"""
=================================================
     Introduction to Machine Learning (67577)
            IML HACKATHON, June 2020

Author(s):

=================================================
"""

import pandas as pd
import pickle
from classifier import final_pre_proc
from strong  import WeakTeam
from binarysearch import binarysearch
from adaboost import AdaBoost
from ex4_tools import DecisionStump

class FlightPredictor:
    def __init__(self, path_to_weather=''):
        """
        Initialize an object from this class.
        @param path_to_weather: The path to a csv file containing weather data.
        """
        #raise NotImplementedError
        self.mod = pickle.load(open("./BinAgent" , "rb"))

    def predict(self, x):
        """
        Recieves a pandas DataFrame of shape (m, 15) with m flight features, and predicts their
        delay at arrival and the main factor for the delay.
        @param x: A pandas DataFrame with shape (m, 15)
        @return: A pandas DataFrame with shape (m, 2) with your prediction
        """

        _dataset, y, x_factor,y_factor = final_pre_proc(x)
        self.mod.predict(x)
```

9 task1/src/models.py

```python
import numpy as np
import pandas as pd
from sklearn.svm import SVC

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import Lasso, Ridge

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import \
    Axes3D  # <--- This is important for 3d plotting
import pandas as pd
from pandas import DataFrame
# from plotnine import *

import matplotlib as mpl
import matplotlib.pyplot as plt

from random import random

from ex4_tools import DecisionStump


class abcModel:

    def __init__(self):
        self.mod = None

    def fit(self, X, y):
        self.mod.fit(self.bias(X), y.ravel())

    def predict(self, X):
        return self.mod.predict(self.bias(X))

    def get_hyperplan(self):
        pass

    def bias(self, X):
        return np.insert(X, 0, 1, 1)

    def draw(self):
        pass

    def score(self, X, y):
        return {
            "num_samples": 0,
            "error" : 0,
            "accuracy" : 0,
            "FPR": 0,
            "TPR": 0,
            "precision": 0,
            "recall": 0
        }


# [v]
class Perceptron(abcModel):
    def __init__(self):
```

```
60          self.W = np.array([])
61          super().__init__()
62
63      def fit(self, _X, y):
64          X = self.bias(_X)
65
66          self.W = np.zeros(shape=X.shape[1])
67
68          def not_classifiy():
69              return [(x, y) for x in X if self.predict(x) != y]
70
71          _updated = True
72          while _updated:
73              _updated = False
74              for i in range(len(y)):
75                  if np.dot(self.W, X[i]) * y[i] <= 0:
76                      self.W = self.W + (X[i] * y[i])
77                      _updated = True
78          return self.W
79
80      def predict(self, X):
81          return np.sign(self.W @ self.bias(X).transpose())
82
83
84  class LDA(abcModel):
85      def __init__(self):
86          super().__init__()
87
88      def fit(self, X, y):
89          X = self.bias(X)
90
91          def gen_delta_y(X, y, y_val):
92              _X = X[y == y_val]
93              lnP = np.log(len(y == y_val) / len(y))
94
95              aritmetic_mean = np.array([np.mean(u) for u in _X.transpose()])
96              _cov = np.cov(X.transpose())
97              _inv_cov = np.linalg.pinv(_cov)
98
99              def delta(x):
100                 return x.transpose() @ _inv_cov @ aritmetic_mean - \
101                        0.5 * aritmetic_mean.transpose() @ _inv_cov @ aritmetic_mean + \
102                        lnP
103
104             return delta
105
106         self.deltas = [gen_delta_y(X, y, y_val) for y_val in [-1, 1]]
107
108     def predict(self, U):
109         return np.array([{0: -1.0, 1: 1.0}[
110                             np.argmax([_delta(u) for _delta in self.deltas])]
111                         for u in self.bias(U)])
112
113
114 class SVM(abcModel):
115     def __init__(self):
116         super().__init__()
117         self.mod = SVC(C=1e10, kernel='linear')
118
119     def coef_(self):
120         return self.mod.coef_
121
122
123 class Logistic(abcModel):
124     def __init__(self):
125         super().__init__()
126         self.mod = LogisticRegression(solver='liblinear')
127
```

```
128     def fit(self, X, y):
129         super().fit(X, y.flatten())
130 
131     # def predict(self, X):
132     #     return self.mod.predict(self.bias(X))
133 
134 class DecisionTree(abcModel):
135     def __init__(self, max_depth=2):
136         super().__init__()
137         self.mod = DecisionTreeClassifier(max_depth=2)
138 
139 
140 class KNearestNeighbor(abcModel):
141 
142     def __init__(self):
143         super().__init__()
144         self.mod = KNeighborsClassifier(n_neighbors=40)
145 
146 
147 class RidgeClassifier(abcModel):
148 
149     def __init__(self):
150         super().__init__()
151         self.mod = Ridge(alpha=0, normalize=True)
152 
153 
154 class LassoClassifier(abcModel):
155 
156     def __init__(self):
157         super().__init__()
158         self.mod = Lasso(alpha=0, normalize=True)
159 
160 class DecisionStumpWarper(abcModel):
161     def __init__(self):
162         super().__init__()
163         self.mod = DecisionStump( )
164 
165     def fit(self, X, y, D=None):
166         self.mod.fit(D, X, y.flatten())
167 
168     def predict(self, X):
169         return self.mod.predict(X)
170 
171 
172 def label_f(weight, bias):
173     def _label_f(x):
174         return np.sign(np.dot(weight, x) + bias)
175 
176     return _label_f
177 
178 
179 def draw_points(m, label_function=label_f(np.array([0.3, -0.5]), 0.1)):
180     def _draw_points_n(m, n):
181         return np.random.multivariate_normal(
182             np.zeros(n), np.identity(n), m)
183 
184     X = _draw_points_n(m, 2)
185     y = np.array([label_function(vec) for vec in X])
186     return X, y
187 
188 
189 def analyze_clssifiers():
190     # def generate_line_prec(prec):
191 
192     def plot(prec, _svm, X, y):
193         plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1])
194         plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1])
195 
```

```
196             min_x, max_x = min(X[:, 0]), max(X[:, 0])
197             min_y, max_y = min(X[:, 1]), max(X[:, 1])
198
199             _min_range, _max_range = min(min_x, min_y), max(max_x, max_y)
200             xx = [_min_range, _max_range]
201
202             def get_y(W, _x):
203                 return -(W[0] + _x * W[1]) / W[2] if W[2] != 0 else -W[0]
204
205             def get_y_prep(_x):
206                 return get_y(prec.W, _x)
207
208             def get_y_svm(_x):
209                 print(_svm.coef_()[0])
210                 return get_y(_svm.coef_()[0], _x)
211
212             def get_true_y(_x):
213                 return 0.1 / 0.5 + 0.3 / 0.5 * _x
214
215             plt.xlim([_min_range, _max_range])
216             plt.ylim([_min_range, _max_range])
217
218             middle = (_min_range + _max_range) / 2
219
220             def print_line(msg, _f, _color):
221                 xx = [_min_range, _max_range]
222                 yy = [_f(_x) for _x in xx]
223                 _x = middle + 2 * (0.5 - random())
224                 _y = _f(_x)
225                 plt.plot(xx, yy, color=_color)
226                 plt.annotate(msg, color=_color,
227                              xy=(_x, _y), xycoords='data',
228                              xytext=(_x + 0.3, _y), textcoords='data',
229                              arrowprops=dict(arrowstyle="->"))
230
231             print_line("prep", get_y_prep, "C5")
232             print_line("svm", get_y_svm, "C4")
233             print_line("true plane", get_true_y, "C2")
234             plt.title("svm vs prep")
235             plt.xlabel("x)")
236             plt.ylabel("y")
237             plt.show()
238
239         for m in [5, 10, 15, 25, 70]:
240             X, y = draw_points(m)
241             blues, reds = X[y == 1], X[y == -1]
242             _modes = [Perceptron(), SVM()]
243             for _model in _modes:
244                 _model.fit(deepcopy(X), y)
245
246             plot(_modes[0], _modes[1], X, y)
247
248
249     def expanded_analyze_clssifiers():
250         times, k = 7, 1000
251         modles = []
252         models_num = 3
253
254         def genrate_real_plane(m):
255             _f = label_f(np.array([random(), random()]), random())
256             X, y = draw_points(m, label_function=_f)
257             while (-1 not in y) or (1 not in y):
258                 X, y = draw_points(m, label_function=_f)
259             return X, y, _f
260
261         def accur(mod, _f, Z):
262             _prob = 0
263             for x, y in zip(map(_f, Z), mod.predict(Z)):
```

```
            if x == y:
                _prob += 1
        return _prob / len(Z)

    def one_iteraion(m):
        _modes = [Perceptron(), SVM(), LDA()]
        X, y, _f = genrate_real_plane(m)
        ret = []
        for _model in _modes:
            print(type(_model))
            _model.fit(deepcopy(X), y)
            Z, _ = draw_points(k)
            ret.append(accur(_model, _f, Z))
        return np.array(ret)

    def calc_mean_performance(M=[5, 10, 15, 25, 70]):
        ret = []
        for m in M:
            _mean = np.zeros(models_num)
            for _ in range(times):
                _mean += one_iteraion(m)
            ret.append(_mean / times)
        return M, np.array(ret)

    m, mean_performance = calc_mean_performance()
    for _model_num, _name in enumerate(["perc", "svm", "lda"]):
        print(_name)
        print(mean_performance)
        plt.plot(m, mean_performance[:, _model_num])
    plt.legend(["perc", "svm", "lda"])
    plt.title("calc_mean_performance")
    plt.xlabel("m (size of the given training data)")
    plt.ylabel("propability of successes")
    plt.show()


if __name__ == "__main__":
    X, y = draw_points(10)
    # p = Perceptron()

    from copy import deepcopy

    models_class = [Perceptron, SVM, Logistic, DecisionTree, LDA]
    models = []
    for mod in models_class:
        models.append(mod())
        print("{} init ".format(type(mod)))

    for mod in models:
        mod.fit(deepcopy(X), y)
        print("{} fit ".format(type(mod)))

    for mod in models:
        print(mod.predict(deepcopy(X)))
        print("{} predict ".format(type(mod)))

    # analyze_clssifiers()
    expanded_analyze_clssifiers()
```

10 task1/src/strong.py

```
from models import abcModel, DecisionTree, Logistic, SVM, DecisionStumpWarper
from adaboost import AdaBoost, AdaBoostList
from itertools import combinations
import numpy as np
import pandas as pd
from binarysearch import binarysearch
from datetime import date
from random import shuffle
import pickle

from classifier import final_pre_proc
from sklearn.model_selection import train_test_split


# class WeakFactory:
#     class WeakLernerByFeature(abcModel):

#         def __init__(self, _model):
#             self.mod = _model

#         def fit(self, X, y):
#             self.mod.fit(X,y)

#         def predict(self, X):
#             return self.mod.predict()


#     def __init__ (self):
#         pass

#     @staticmethod
#     def CreateWeaks(self):

#         return {

#                 "DayOfWeek" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "FlightDate" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Reporting_Airline" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Tail_Number" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Flight_Number_Reporting_Airline" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Origin" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "OriginCityName" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "OriginState" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Dest" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "DestCityName" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "DestState" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "CRSDepTime" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "CRSArrTime" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "CRSElapsedTime" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Distance" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "ArrDelay" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "DelayFacto" : WeakFactory ( DecisionTree(max_depth=2) )


    #return generateWeakClass( DecisionTree )
    # return generateWeakClass( Logistic )

# " extracrd out for pickiling "
class WeakTeam(DecisionStumpWarper):
```

```
60
61      def __init__(self, featuers=True):
62          DecisionStumpWarper.__init__(self)
63          self.featuers = featuers
64
65      def filterX( self, X ):
66          ret = np.array( pd.DataFrame( { featuer:  X[featuer] for featuer in self.featuers  }))
67          return ret
68
69      def train(self, X, y, D=None):
70          #print(f"[@] train on features : { self.featuers}")
71
72          if D is None:
73              super().fit(
74                  np.array(self.filterX(X)),  y)
75          else:
76              super().fit(
77                  np.array(self.filterX(X)), y, D)
78
79      def predict(self, X):
80          return super().predict( self.filterX(X) )
81
82  def generateTeamClass(featuers):
83      return (WeakTeam, featuers)
84
85
86  #
87
88
89  # def binarysearch_read(_file):
90  #     mods = {}
91  #     for banch in re.split('$', _file.read()):
92  #         treshold, _strmod = re.split(':', banch)
93  #         mods[float(treshold)] = AdaBoost_read(banch)
94
95
96  def calc_error(model, _dataframe, y, agents):
97      _error = 0
98      z = (y.flatten() - 0.5 * np.ones(len(y))) * 2
99      for _bool in (model.predict(_dataframe, max_t=agents) != z):
100         _error += {False: 0, True: 1}[_bool]
101     return _error / len(y)
102
103
104 import heapq
105
106
107 def hash_strings(featuers, _list):
108     ret = 0
109     base = 1
110     for featuer in featuers:
111         if featuer in _list:
112             ret += base
113         base *= 10
114     return ret
115
116
117 def learn(_dataframe, y, featuers, teams=set(), depth=5, orignal=[],
118           _hased=set()):
119     if len(teams) == 0:
120         teams = [[featuer] for featuer in featuers]
121
122     agents = 1
123     _hashed = set()
124     new_team = []
125
126     def create_subgroups():
127         subgroups = []
```

```
        for featuer in featuers:
            for team in teams:
                if featuer not in team:
                    _hash = hash_strings(orignal, team + [featuer])
                    if _hash not in _hashed:
                        # print(f"i was here {team + [featuer]}, _hash : {_hash}")
                        subgroups.append(generateTeamClass(team + [featuer]))
                        _hashed.add(_hash)
        return subgroups

    strongGroups = []
    heap = []

    for weak in create_subgroups():
        _model = AdaBoost(weak, agents, support_wights=True)
        _model.train(_dataframe, y)
        strongGroups.append(_model)
        heapq.heappush(heap, (
            -calc_error(_model, _dataframe, y, agents), _model.h[0].featuers,
            _model))


        if len(heap) > 3:
            heapq.heappop( heap )

    if depth == 0:
        while len(heap) > 1:
            train_error, _featuers, _model =  heapq.heappop( heap )
        train_error, _featuers, _model =  heapq.heappop( heap )
        return train_error, _featuers, _model

    else:
        teams = []
        while len(heap) > 1:

            heapq.heappop( heap )
            train_error, _featuers, _model =  heapq.heappop( heap )
            teams.append( _featuers  )
        return learn(_dataframe, y, featuers, teams, depth - 1,
                     orignal=orignal)


def generateY(_dataset, treshold=0):
    y = []
    for _bool in _dataset["ArrDelay"] > treshold:
        y.append({False: [0], True: [1]}[_bool])
    return np.array(y)


def generateYbyString(_dataset, s, treshold=0):
    y = []
    for _bool in _dataset[s] > treshold:
        y.append({False: [0], True: [1]}[_bool])
    return np.array(y)


def pairs():
    df1 = df_copy.loc[np.logical_or(df_copy["DelayFactor"] == 0,
                                    df_copy["DelayFactor"] == 1)]
    df2 = df1.drop(columns=["DelayFactor"])
    _, _, classifier1 = learn(df2, df1["DelayFactor"].as_matrix(), df2.keys(), orignal=df2.keys())

    df1 = df_copy.loc[np.logical_or(df_copy["DelayFactor"] == 1,
                                    df_copy["DelayFactor"] == 2)]
    df2 = df1.drop(columns=["DelayFactor"])
    _, _, classifier2 = learn(df2, df1["DelayFactor"].as_matrix() - 1, df2.keys(),
          orignal=df2.keys())

```

```
    df1 = df_copy.loc[np.logical_or(df_copy["DelayFactor"] == 2,
                                    df_copy["DelayFactor"] == 3)]
    df2 = df1.drop(columns=["DelayFactor"])
    _, _, classifier3 = learn(df2, df1["DelayFactor"].as_matrix() - 2, df2.keys(),
          orignal=df2.keys())

    df1 = df_copy.loc[np.logical_or(df_copy["DelayFactor"] == 3,
                                    df_copy["DelayFactor"] == 1)]
    df2 = df1.drop(columns=["DelayFactor"])
    s = df1["DelayFactor"].as_matrix()
    s[s == 3] = 0
    df1 = df1.drop(columns=["DelayFactor"])
    df1["DelayFactor"] = s
    _, _, classifier4 = learn(df2, df1["DelayFactor"].as_matrix(), df2.keys(),
          orignal=df2.keys())

    df1 = df_copy.loc[np.logical_or(df_copy["DelayFactor"] == 2,
                                    df_copy["DelayFactor"] == 0)]
    df2 = df1.drop(columns=["DelayFactor"])
    s = df1["DelayFactor"].as_matrix()
    s[s == 2] = 1
    df1 = df1.drop(columns=["DelayFactor"])
    df1["DelayFactor"] = s
    _, _, classifier5 = learn(df2, df1["DelayFactor"].as_matrix(), df2.keys(),
          orignal=df2.keys())

    df1 = df_copy.loc[np.logical_or(df_copy["DelayFactor"] == 3,
                                    df_copy["DelayFactor"] == 0)]
    df2 = df1.drop(columns=["DelayFactor"])
    s = df1["DelayFactor"].as_matrix()
    s[s == 3] = 1
    df1 = df1.drop(columns=["DelayFactor"])
    df1["DelayFactor"] = s
    _, _, classifier6 = learn(df2, df1["DelayFactor"].as_matrix(), df2.keys(),
          orignal=df2.keys())
    classifier1.predict(df2)
    return [classifier1, classifier2, classifier3, classifier4, classifier5, classifier6]



def pre_proc(_dataset, droped_fe, categorical ):
    y = generateY(_dataset)

    if len(categorical) > 0:
        cat = pd.DataFrame(
            pd.get_dummies(_dataset[categorical].astype('category')))

    _dataset = _dataset.drop(droped_fe + categorical, axis=1)
    if len(categorical) > 0:
        _dataset_prepoc = pd.concat(
            [_dataset.reset_index(drop=True), cat.reset_index(drop=True)],
            axis=1)
    else:
        _dataset_prepoc = _dataset

    print(_dataset_prepoc)
    return _dataset_prepoc, y


featuers = ["DayOfWeek",
            "FlightDate",
            "Reporting_Airline",
            "Tail_Number",
            "Flight_Number_Reporting_Airline",
            "Origin",
            "OriginCityName",
            "OriginState",
```

```
            "Dest",
            "DestCityName",
            "DestState",
            "CRSDepTime",
            "CRSArrTime",
            "CRSElapsedTime",
            "Distance"]

droped_fe = ['Flight_Number_Reporting_Airline',
             'Tail_Number',
             #   'DayOfWeek',
             'FlightDate',
             'ArrDelay',
             'DelayFactor']

categorical = [
    'OriginCityName'
    , 'OriginState'
    , 'Origin'
    , 'Dest'
    , 'DestCityName'
    , 'DestState'
    , 'Reporting_Airline']



if __name__ == "__main__" :
    original_dataset = pd.read_csv("~/data/train_data.csv", nrows=80)

    print("[#] before pre processing")
    print(original_dataset)
    _mods = {}
    _minrange, _maxrange = -30, 30
    _dataset, y, x_factor,y_factor = final_pre_proc(original_dataset)

    df_copy = _dataset.copy()
    df_copy["DelayFactor"] = y_factor
    pairs()

    start_range , end_range = np.ones(len(y)) * _minrange , np.ones(len(y)) * _maxrange
    for i, time in enumerate( np.arange(_minrange, _maxrange,  (_maxrange-_minrange)/ 2 ) ):
        train_error, _featuers, _mods[time] = learn(_dataset,
                                                    generateY(original_dataset,
                                                              time),
                                                    _dataset.keys(), teams=[
                ['CRSDepTime', 'DayOfWeek']], orignal=_dataset.keys())
        print(f"{time} : {_featuers} : {train_error}")

    Bagent = binarysearch( _mods )
    # _middles = Bagent.predict( _dataset , start_range , end_range)
    # print(_middles)

    _dataset, y, x_factor, y_factor = final_pre_proc(
        pd.read_csv("~/data/train_data.csv", nrows=10000)[9800:])
    # _dataset, y = pre_proc(original_dataset, droped_fe , categorical )
    _middles = Bagent.mods[0.0].predict(_dataset)
    _middles[_middles > 0] = 1
    t = sum((_middles - y.flatten()) ** 2 / len(y))
    print(f"[error]: {t}")
    # print( )

    # _dataset, y, x_factor,y_factor= final_pre_proc( pd.read_csv("~/data/train_data.csv", nrows=10000 )[9800:] )
    # _middles = Bagent.mods[0.0].predict( _dataset )
    # _middles[ _middles > 0 ] = 1
    # t = sum( (_middles- y.flatten())**2/len(y))
    # print (  f"[error]: {t}"  )

    with open("./BinAgent", "wb") as f:
```

```
332             pickle.dump(Bagent, f)
333
334
335
336
337     # class StrongClassifer:
338
339     #     def __init__ (self, features):
340     #         pass
341
342     #     def predict(self, X):
343     #         pass
344
345     #     def train(self, X, y):
346     #         pass
```

11 task1/src/strong2.py

```python
from HUJIHACK.Task1.source.models import abcModel, DecisionTree
from HUJIHACK.Task1.source.adaboost import AdaBoost, AdaBoostList
from itertools import combinations
import numpy as np
import pandas as pd
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
from datetime import date
from sklearn.linear_model import LogisticRegression

import pandas as pd
import numpy as np
import seaborn as sns
import math

from sklearn.linear_model import Ridge

from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error


# class WeakFactory:
#     class WeakLernerByFeature(abcModel):

#         def __init__(self, _model):
#             self.mod = _model

#         def fit(self, X, y):
#             self.mod.fit(X,y)

#         def predict(self, X):
#             return self.mod.predict()


#     def __init__ (self):
#         pass

#     @staticmethod
#     def CreateWeaks(self):

#         return {

#                 "DayOfWeek" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "FlightDate" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Reporting_Airline" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Tail_Number" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Flight_Number_Reporting_Airline" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Origin" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "OriginCityName" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "OriginState" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Dest" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "DestCityName" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "DestState" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "CRSDepTime" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "CRSArrTime" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "CRSElapsedTime" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "Distance" : WeakFactory ( DecisionTree(max_depth=2) ),
#                 "ArrDelay" : WeakFactory ( DecisionTree(max_depth=2) ),
```

```
#                 "DelayFacto" : WeakFactory ( DecisionTree(max_depth=2) )
#         }

def generateTeamClass(featuers):
    class WeakTeam(DecisionTree):

        def __init__(self):
            DecisionTree.__init__(self, max_depth=3)
            self.featuers = featuers

        def filterX(self, X):
            ret = np.array(pd.DataFrame({featuer: X[featuer] for featuer in self.featuers}))
            return ret

        def train(self, X, y):
            print(f"[@] train on features : {self.featuers}")
            super().fit(
                np.array(self.filterX(X)), y)

        def predict(self, X):
            return super().predict(self.filterX(X))

    return WeakTeam


def pre_proc_new(_dataset, droped_fe, categorical):
    y_del = []
    y_factor = []
    for delay, factor in zip(_dataset["ArrDelay"], _dataset["DelayFactor"]):
        y_del.append(delay)
        y_factor.append(factor)
    y_del = np.array(y_del)
    y_factor = np.array(y_factor)
    for index, row in _dataset.iterrows():
        _dataset.loc[index, 'CRSElapsedTime'] = math.floor(row['CRSElapsedTime'] / 10)
        _dataset.loc[index, 'CRSArrTime'] = math.floor(row['CRSArrTime'] / 100)
        _dataset.loc[index, 'CRSDepTime'] = math.floor(row['CRSDepTime'] / 100)
    cat = pd.DataFrame(pd.get_dummies(_dataset[categorical].astype('category')))
    _dataset = _dataset.drop(droped_fe + categorical, axis=1)
    _dataset_prepoc = pd.concat([_dataset.reset_index(drop=True), cat.reset_index(drop=True)], axis=1)
    return _dataset_prepoc, y_del, y_factor


def ridge_reg(x, y):
    # ridge = Ridge()
    # parameters = {'alpha': [1, 5, 10, 20, 25, 30, 40, 50, 60, 70, 1000]}
    # ridge_regressor = GridSearchCV(ridge, parameters, cv=5)
    # ridge_regressor.fit(x, y)
    # b = ridge_regressor.predict(x)
    # print(ridge_regressor.best_params_)
    # print(ridge_regressor.best_score_)
    # a = range(1,10001)
    # plt.scatter(a, y, c='b', s=1, alpha=0.6, label='Training data')
    # plt.scatter(a, b, c='red', s=1, label='prediction')
    # # plt.plot(a, y, 'k', label='True function')
    # plt.legend()
    # plt.show()
    # rr = Ridge(alpha=5)
    rr = LogisticRegression(solver='liblinear')
    rr.fit(x, y)
    pred_train_rr = rr.predict(x)
    print(rr.score(x, y))
    # print(np.sqrt(mean_squared_error(y, pred_train_rr)))
    # print(r2_score(y, pred_train_rr))

    # pred_test_rr = rr.predict(X_test)
    # print(np.sqrt(mean_squared_error(y_test, pred_test_rr)))
    # print(r2_score(y_test, pred_test_rr))
```

```
128
129
130 def identify_corelated_features(df):
131     corr = df.corr()
132     print(sns.heatmap(corr))
133     columns = np.full((corr.shape[0],), True, dtype=bool)
134     for i in range(corr.shape[0]):
135         for j in range(i + 1, corr.shape[0]):
136             if corr.iloc[i, j] >= 0.9:
137                 if columns[j]:
138                     columns[j] = False
139     selected_columns = df.columns[columns]
140     df = df[selected_columns]
141     return df
142
143
144 def learn(_dataframe, y, featuers):
145     agents = 3
146     group_size = 1
147     subgroups = [generateTeamClass(team) for team in combinations(featuers, group_size)]
148
149     def calc_error(model, _dataframe, y):
150         _error = 0
151         for _bool in (model.predict(_dataframe, max_t=1) != y).flatten():
152             _error += {False: 0, True: 1}[_bool]
153         return _error / len(y)
154
155     strongGroups = []
156     for weak in subgroups:
157         _model = AdaBoost(weak, agents)
158         _model.train(_dataframe, y)
159         strongGroups.append(_model)
160
161     return strongGroups[np.argmin([calc_error(_model, _dataframe, y) for _model in strongGroups])]
162
163
164 if __name__ == "__main__":
165     _dataset = pd.read_csv("../data/train_data.csv", nrows=10000)
166
167
168     # categorical = ['view' , 'waterfront', 'bedrooms', 'grade', 'floors', 'condition', 'bathrooms']
169     # cat = pd.DataFrame ( { 'id' : _dataset_prepoc['id'] } , pd.get_dummies(_dataset_prepoc[categorical].astype('category')
170
171     def pre_proc(_dataset, droped_fe, categorical):
172
173         def generateY(_dataset):
174             y = []
175             for _bool in _dataset["ArrDelay"] > 0:
176                 y.append({False: [0], True: [1]}[_bool])
177             return np.array(y)
178
179         # def remove_end_cases(_frame):
180         #     return _frame[  _frame['price'] > 0  ]
181         y = generateY(_dataset)
182
183         cat = pd.DataFrame(pd.get_dummies(_dataset[categorical].astype('category')))
184         _dataset = _dataset.drop(droped_fe + categorical, axis=1)
185         _dataset_prepoc = pd.concat([_dataset.reset_index(drop=True), cat.reset_index(drop=True)], axis=1)
186         return _dataset_prepoc, y
187
188
189     featuers = ["DayOfWeek",
190                 "FlightDate",
191                 "Reporting_Airline",
192                 "Tail_Number",
193                 "Flight_Number_Reporting_Airline",
194                 "Origin",
195                 "OriginCityName",
```

```
                "OriginState",
                "Dest",
                "DestCityName",
                "DestState",
                "CRSDepTime",
                "CRSArrTime",
                "CRSElapsedTime",
                "Distance"]

    droped_fe = ['Flight_Number_Reporting_Airline',
                 'Tail_Number',
                 'FlightDate',
                 'ArrDelay',
                 'DelayFactor']

    droped_fe_new = ['Flight_Number_Reporting_Airline',
                     'Tail_Number',
                     'FlightDate',
                     'ArrDelay',
                     'DelayFactor',
                     "OriginCityName",
                     "OriginState",
                     "DestCityName",
                     "DestState", ]

    categorical_new = ['Reporting_Airline', 'Origin', 'Dest']

    categorical = [
        'OriginCityName'
        , 'OriginState'
        , 'Origin'
        , 'Dest'
        , 'DestCityName'
        , 'DestState'
        , 'Reporting_Airline']

    # "ArrDelay",
    # "DelayFactor"]

    # print("[#] before pre processing")
    # print(_dataset)
    # _dataset, y = pre_proc(_dataset, droped_fe, categorical)
    # print("[#] after pre processing")
    # print(_dataset)
    # print("[#] y' vector ")
    # print(y)
    #
    # print(_dataset.keys())
    # _mod = learn(_dataset, y, _dataset.keys())
    # print("[#] best featuers:")
    # print(_mod.h[0].featuers)
    x, y_del, y_factor = pre_proc_new(_dataset, droped_fe_new, categorical_new)
    print(x)
    x = identify_corelated_features(x)
    print(x)
    # print(x)
    # print(x.shape)
    # print(y_del)
    # print(y_del.shape)
    # print(y_factor)
    # ridge_reg(x, y_del)
    # print(x)
    # print("after corr\n")
    # print(identify_corelated_features(x,y_del))

# class StrongClassifer:

#     def __init__ (self, features):
```

```
264     #         pass
265
266     #     def predict(self, X):
267     #         pass
268
269     #     def train(self, X, y):
270     #         pass
```

12 task1/src/testmodel.py

```
1
2
3  from model import FlightPredictor
4  import pandas as pd
5  from classifier import final_pre_proc
6  from strong  import WeakTeam
7  from binarysearch import binarysearch
8  from adaboost import AdaBoost
9  from ex4_tools import DecisionStump
10
11 if __name__ == "__main__":
12     print("[@] create FlightPredictor")
13     fp = FlightPredictor( )
14     print("[@] predict")
15     ret = fp.predict( pd.read_csv("~/data/train_data.csv", nrows=10000)  )
16     print(ret)
```

13 task1/src/weather.py

```
import json
import pandas as pd
import pickle


if __name__ == "__main__" :
    df = pd.read_csv("~/data/all_weather_data.csv")
    _json = { }
    for index, row in df.iterrows():
        for key in df.keys():
            _json[ row['day'] ] = {}
            if key != "day":
                _json[ row['day'] ][key] = row[key]
    json.dump( _json ,  open("./wether.json", "w") )


```

14 task1/src/wether.json

```
1  {"01-01-10": {"max_wind_gust_kts": "0"}, "02-01-10": {"max_wind_gust_kts": "0"}, "03-01-10": {"max_wind_gust_kts": "0"}, "04
```