{¥rtf1¥ansi¥ansicpg1252¥cocoartf2636
¥cocoatextscaling0¥cocoaplatform0{¥fonttbl¥f0¥fswiss¥fcharset0 Helvetica;}
{¥colortbl;¥red255¥green255¥blue255;¥red255¥green255¥blue255;}
{¥*¥expandedcolortbl;;¥cssrgb¥c99989¥c100000¥c99983;}
¥margl1440¥margr1440¥vieww11520¥viewh8400¥viewkind0
¥pard¥tx560¥tx1121¥tx1681¥tx2242¥tx2803¥tx3363¥tx3924¥tx4485¥tx5045¥tx5606¥tx6166¥tx6727¥tx7288¥tx78
48¥tx8409¥tx8970¥tx9530¥tx10091¥tx10651¥tx11212¥tx11773¥tx12333¥tx12894¥tx13455¥tx14015¥tx14576¥tx1
5136¥tx15697¥tx16258¥tx16818¥tx17379¥tx17940¥tx18500¥tx19061¥tx19621¥tx20182¥tx20743¥tx21303¥tx2186
4¥tx22425¥tx22985¥tx23546¥tx24106¥tx24667¥tx25228¥tx25788¥tx26349¥tx26910¥tx27470¥tx28031¥tx28591¥t
x29152¥tx29713¥tx30273¥tx30834¥tx31395¥tx31955¥tx32516¥tx33076¥tx33637¥tx34198¥tx34758¥tx35319¥tx35
880¥tx36440¥tx37001¥tx37561¥tx38122¥tx38683¥tx39243¥tx39804¥tx40365¥tx40925¥tx41486¥tx42046¥tx42607
¥tx43168¥tx43728¥tx44289¥tx44850¥tx45410¥tx45971¥tx46531¥tx47092¥tx47653¥tx48213¥tx48774¥tx49335¥tx
49895¥tx50456¥tx51016¥tx51577¥tx52138¥tx52698¥tx53259¥tx53820¥tx54380¥tx54941¥tx55501¥tx56062¥sllea
ding20¥pardirnatural¥partightenfactor0

¥f0¥fs24 ¥cf2 # %%¥
import numpy as np¥
import pandas as pd¥
import matplotlib.pyplot as plt¥
from itertools import combinations, permutations¥
from scipy.optimize import minimize, root¥
from tqdm import tqdm¥
from joblib import Parallel, delayed¥
from collections import defaultdict¥
from collections.abc import Iterable¥
¥
import warnings¥
warnings.simplefilter('ignore')¥
¥
%%¥
class ApproxRUM():¥
    """¥
    Attributes¥
    ----------¥
        K : int¥
            dimension of features¥
        X : tuple¥
            like (0, 1, ..., K-1)¥
        cal_D : list of tuples¥
            list of choice sets¥
    """¥
    def __init__(self, features):¥
        """¥
        Parameters¥
        ----------¥
            features : array¥
                features of each alternative¥
        """¥
        self.features = features¥
        self.K = self.features.shape[1]¥
        self.n = self.features.shape[0]¥
        self.X = tuple(range(self.n))¥
        #self.cal_D = list(combinations(self.X, 2)) + [self.X] + list(combinations(self.X, 3))¥
        self.cal_D = sum([list(combinations(self.X, i)) for i in range(1, self.n + 1)], [])¥
        #self.cal_D = [tuple(D) for D in self.cal_D]¥
        self.num_choice_sets = len(self.cal_D)¥
¥

```
    def logit(self, D, beta, fixed_effects=0):¥
        """¥
        choice probabilities based on a logit model¥
¥
        Parameters¥
        ----------¥
            D : tuple¥
                a choice set that contains X_ALT¥
            beta : K-dim array¥
                coefficients of logit model¥
            fixed_effects : len(D)-dim array, default 0¥
                fixed effects¥
¥
        Returns¥
        -------¥
            probs : array¥
                an np.array of probabilities with length len(D)¥
        """¥
        num_alts = len(D)¥
        K = len(beta)¥
        D_feat = self.features[tuple([D])]¥
        assert D_feat.shape == (num_alts, K)¥
        lin_part_vec = D_feat @ beta + fixed_effects¥
        lin_part_vec_normalized = lin_part_vec - max(lin_part_vec)¥
        pre_probs = np.exp(lin_part_vec_normalized)¥
        probs = pre_probs / pre_probs.sum()¥
        return probs¥
¥
    def mixed_logit(self, D, beta_mat, lambda_vec, fixed_effects=0):¥
        """¥
        choice probabilities based on a mixed logit model¥
¥
        Parameters¥
        ----------¥
            D : tuple¥
                choice set¥
            beta_mat : (M, K)-dim array¥
                matrix of coeeficients¥
            lambda_vec : M-dim array¥
                M-dimensional vector of mixture weights¥
            fixed_effects : len(D)-dim array, default 0¥
                fixed effects¥
¥
        Returns¥
        -------¥
            probs : array¥
                an np.array of probabilities with length len(D)¥
        """¥
        probs = 0¥
        for m, lam in enumerate(lambda_vec):¥
            beta = beta_mat[m, :]¥
            probs += lam * self.logit(D, beta, fixed_effects)¥
        return probs¥
¥
    def get_gamma(self, D, lambda_old, beta_old_mat):¥
        """¥
        Parameters¥
```

```
        ----------¥
            lambda_old : M-dim array¥
                previous mixture weights¥
            beta_old_mat : (M, K)-dim array¥
                previous coefficients¥
¥
        Returns¥
        -------¥
            gamma_mat : (M, len(D))-dim array¥
                an np.array of gammas with shape M ¥¥times len(D)¥
        """¥
        prob_vec = np.array([self.logit(D, beta) for beta in beta_old_mat])  # M ¥¥times len(D)¥
        weighted_prob_vec = prob_vec * lambda_old.reshape(-1, 1)  # M ¥¥times len(D)¥
        gamma_mat = weighted_prob_vec / weighted_prob_vec.sum(axis=0)   # M ¥¥times len(D)¥
        return gamma_mat¥
¥
    def get_lambda(self, gamma_new):¥
        """¥
        Parameters¥
        ----------¥
            gamma_new : list¥
                list of np.arrays in order of cal_D¥
¥
        Returns¥
        -------¥
            lambda_vec : M-dim array¥
                an np.array of lambdas with length M¥
        """¥
        pre_lambda = sum([(gamma_new[i] * self.rho_set[D].reshape(1, -1)).sum(axis=1) for i, D in
enumerate(self.cal_D)])¥
        lambda_vec = pre_lambda / self.num_choice_sets¥
        return lambda_vec¥
¥
    def _get_beta_mat(self, gamma_new, beta_initial_mat):¥
        """¥
        Find the optimal beta_mat naively; slow¥
¥
        Parameters¥
        ----------¥
            gamma_new : list¥
                list of np.arrays in order of cal_D¥
            beta_initial_mat : (M, K)-dim array¥
                initial valus of beta¥
¥
        Returns¥
        -------¥
            beta_new_mat : (M, K)-dim array¥
                an np.array with shape M ¥¥times K¥
¥
        Notes¥
        -----¥
            This is no longer used. See get_beta_mat, which returns same outcomes but is much faster than this.¥
        """¥
        M = len(gamma_new[0])¥
        #fix m¥
        def ll(beta_m, m):¥
            ll_m = 0¥
```

```
        for i, D in enumerate(self.cal_D):¥
            gamma_new_m = gamma_new[i][m, :]  # len(D)¥
            lin_part_vec = self.features[tuple([D])] @ beta_m  # len(D)¥
            denom_logit = np.sum(np.exp(lin_part_vec))  # scalor¥
            log_denom_logit = np.log(denom_logit)  # scalor¥
            ll_m += sum(self.rho_set[D] * gamma_new_m * (lin_part_vec - log_denom_logit))¥
        return ll_m¥
¥
    beta_new_mat = []¥
    ll_list = []¥
    for m in range(M):¥
        target_func = lambda beta_m: -ll(beta_m, m)¥
        # optimize target_func wrt beta_m¥
        method = "BFGS"¥
        res = minimize(target_func, beta_initial_mat[m, :], bounds=[(-20, 20) for _ in range(self.K)],
method=method)¥
        # let beta_m_new the solution¥
        if res.success:¥
            beta_m_new = res.x¥
        else:¥
            beta_m_new = beta_initial_mat[m, :]¥
        beta_new_mat.append(beta_m_new)¥
        ll_list.append(res.fun)¥
¥
    beta_new_mat = np.array(beta_new_mat)  # M ¥¥times K¥
    return beta_new_mat¥
¥
  def get_beta_mat(self, gamma_new, beta_initial_mat):¥
    """¥
    Find the optimal beta_mat based on the FOC; fast¥
¥
    Parameters¥
    ----------¥
        gamma_new : list¥
            list of np.arrays in order of cal_D¥
        beta_initial_mat : (M, K)-dim array¥
            initial valus of beta¥
¥
    Returns¥
    -------¥
        beta_new_mat : (M, K)-dim array¥
            an np.array with shape M ¥¥times K¥
    """¥
    def eq(beta_m, m):¥
        total = 0¥
        for i, D in enumerate(self.cal_D):¥
            self.rho_set[D]¥
            gamma_new_m = gamma_new[i][m, :]¥
            nomin = (np.exp(self.features[tuple([D])] @ beta_m).reshape(-1, 1) *
self.features[tuple([D])]).sum(axis=0)  # len(D) ¥¥times K¥
            denom = np.exp(self.features[tuple([D])] @ beta_m).sum()¥
            total += sum((self.rho_set[D] * gamma_new_m).reshape(-1, 1) * (self.features[tuple([D])] - nomin /
denom))¥
        return total¥
¥
    M = len(gamma_new[0])¥
    beta_new_mat = []¥
```

```
        for m in range(M):¥
            target_func = lambda beta_m: eq(beta_m, m)¥
            res = root(target_func, beta_initial_mat[m, :])¥
            if res.success:¥
                beta_m_new = res.x¥
            else:¥
                beta_m_new = beta_initial_mat[m, :]¥
            beta_new_mat.append(beta_m_new)¥
¥
        beta_new_mat = np.array(beta_new_mat)  # M ¥¥times K¥
        return beta_new_mat¥
¥
    def EM_algorithm_from_pref(self, target_pref, M, max_iter_num=100):¥
        """¥
        Parameters¥
        ----------¥
            target_pref : tuple¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
        """¥
        rho_est = self.get_rho_from_pref(target_pref)¥
        self.EM_algorithm(rho_est, M, max_iter_num)¥
¥
    def EM_algorithm(self, rho_set, M, max_iter_num=100):¥
        """¥
        Parameters¥
        ----------¥
            rho_set : dict¥
                key : D, value : prob vector (list)¥
            M : int¥
                the number of mixtures¥
            max_iter_num : int¥
                number of iterarions of the EM algorithm¥
¥
        Notes¥
        -----¥
            Fixed effects are not considered in this function.¥
        """¥
        # set the target preference¥
        self.rho_set = rho_set¥
        self.rho = dict(sum([[((D, x), prob) for x, prob in zip(D, probs)] for D, probs in rho_set.items()], []))¥
¥
        # initialize parameters¥
        lambda_old = np.zeros(M) + 1 / M¥
        beta_old_mat = np.random.normal(scale=0.5, size=(M, self.K))¥
¥
        # place to store parameters¥
        self.gamma_list = [None]¥
        self.beta_mat_list = [beta_old_mat]¥
        self.lambda_list = [lambda_old]¥
        self.rho_list = [None]¥
        self.l2_error_list = [np.inf]¥
¥
        for _ in range(max_iter_num):¥
            # E step¥
            gamma_new = [self.get_gamma(D, lambda_old, beta_old_mat) for D in self.cal_D]  # len(cal_D) ¥¥times
(M ¥¥times len(D))¥
¥
```

```
        # M step¥
        lambda_new = self.get_lambda(gamma_new)¥
        beta_initial_mat = beta_old_mat¥
        beta_new_mat = self.get_beta_mat(gamma_new, beta_initial_mat)¥
¥
        # progress just for log¥
        rho_est = self.get_rho_from_mixed_logit(lambda_new, beta_new_mat)¥
        error = self.l2_distance(self.rho, rho_est)¥
¥
        # store parameter records¥
        self.gamma_list.append(gamma_new)¥
        self.beta_mat_list.append(beta_new_mat)¥
        self.lambda_list.append(lambda_new)¥
        self.rho_list.append(rho_est)¥
        self.l2_error_list.append(error)¥
¥
        # termination judgement¥
        if abs(error - self.l2_error_list[-2]) < 1e-6:¥
            break¥
¥
        # update parameters¥
        lambda_old = lambda_new¥
        beta_old_mat = beta_new_mat¥
¥
    def get_rho_from_pref(self, target_pref):¥
        """¥
        Parameters¥
        ----------¥
            target_pref : tuple¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
        """¥
        self.rho = dict()¥
        rank_array = np.argsort(target_pref)¥
        for D in self.cal_D:¥
            rho = ¥{¥}¥
            min_rank = rank_array[tuple([D])].min()¥
            for x in D:¥
                self.rho[(D, x)] = 1 if rank_array[x] == min_rank else 0¥
        rho_set = dict([(D, np.array([self.rho[(D, x)] for x in D])) for D in self.cal_D])¥
        return rho_set¥
¥
    def set_rho_from_pref(self, target_pref):¥
        """¥
        Parameters¥
        ----------¥
            target_pref : tuple¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
        """¥
        self.rho_set = self.get_rho_from_pref(target_pref)¥
¥
    def l2_distance(self, rho_1, rho_2):¥
        """¥
        Parameters¥
        ----------¥
            rho_1 : dict¥
                a stochastic choice function¥
            rho_2 : dict¥
```

```
            a stochastic choice function¥
¥
        Returns¥
        -------¥
            error : float¥
                the l2 distance between rho_1 and rho_2¥
        """¥
        count = 0¥
        squered_error_total = 0¥
        for D in self.cal_D:¥
            for x in D:¥
                count += 1¥
                squered_error_total += (rho_1[(D, x)] - rho_2[(D, x)]) ** 2¥
        error = (squered_error_total / count) ** 0.5¥
        return error¥
¥
    def get_rho_from_logit(self, beta, fixed_effects=0):¥
        """¥
        Parameters¥
        ----------¥
            beta : K-dim array¥
                coefficients¥
            fixed_effects : len(X)-dim array, default 0¥
                fixed effects¥
¥
        Returns¥
        -------¥
            rho_est : dict¥
                a stochastic choice function determined by a logit model with beta¥
        """¥
        if not isinstance(fixed_effects, Iterable):¥
            fixed_effects = np.zeros(len(self.X))¥
        rho_est = dict()¥
        for D in self.cal_D:¥
            probs = self.logit(D, beta, fixed_effects[[D]])¥
            for i, x in enumerate(D):¥
                rho_est[(D, x)] = probs[i]¥
        return rho_est¥
¥
    def get_rho_from_mixed_logit(self, lambda_vec, beta_mat, fixed_effects=0):¥
        """¥
        Parameters¥
        ----------¥
            lambda_vec : M-dim array¥
                mixture weights¥
            beta_mat : (M, K)-dim array¥
                matrix of coefficients¥
            fixed_effects : len(X)-dim array, default 0¥
                fixed effects¥
¥
        Returns¥
        -------¥
            rho_est : dict¥
                a stochastic choice function determined by a logit model with beta¥
        """¥
        if not isinstance(fixed_effects, Iterable):¥
            fixed_effects = np.zeros(len(self.X))¥
```

```
        rho_est = dict()¥
        for D in self.cal_D:¥
            probs = self.mixed_logit(D, beta_mat, lambda_vec, fixed_effects[[D]])¥
            for i, x in enumerate(D):¥
                rho_est[(D, x)] = probs[i]¥
        return rho_est¥
¥
    def set_rho_from_pref_mixture(self, pref_1, pref_2, weight):¥
        """¥
        Parameters¥
        ----------¥
            pref_1 : tuple¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
            pref_2 : tuple¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
            weight : float¥
                weight on pref_1¥
        """¥
        self.rho = defaultdict(int)¥
        rank_array_1 = np.argsort(pref_1)¥
        rank_array_2 = np.argsort(pref_2)¥
        for D in self.cal_D:¥
            min_rank_1 = rank_array_1[tuple([D])].min()¥
            min_rank_2 = rank_array_2[tuple([D])].min()¥
            for x in D:¥
                if rank_array_1[x] == min_rank_1:¥
                    self.rho[(D, x)] += weight¥
                if rank_array_2[x] == min_rank_2:¥
                    self.rho[(D, x)] += 1 - weight¥
        self.rho_set = dict([(D, np.array([self.rho[(D, x)] for x in D])) for D in self.cal_D])¥
¥
    def greedy_algorithm_from_pref(self, target_pref, iter_num=10, target_pref_2=None, weight=None,
fixed_effects=0):¥
        """¥
        Parameters¥
        ----------¥
            target_pref : tuple, default None¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
            target_pref_2 : tuple, default None¥
                a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
            weight : float, default None¥
                weight on target_pref¥
        """¥
        if target_pref_2 and weight:¥
            self.set_rho_from_pref_mixture(target_pref, target_pref_2, weight)¥
        else:¥
            self.set_rho_from_pref(target_pref)¥
        self.greedy_algorithm(self.rho_set, iter_num=iter_num, fixed_effects=fixed_effects)¥
¥
    def greedy_algorithm(self, rho_set, iter_num=10, fixed_effects=0):¥
        """¥
        Haoge's greedy algorithm¥
¥
        Parameters¥
        ----------¥
            rho_set : dict¥
                key : D, value : prob vector (list)¥
```

```
        iter_num : int¥
            number of iterations¥
        fixed_effects : len(X)-dim array, default 0¥
            fixed effects¥
¥
    Returns¥
    -------¥
        error: L2 error¥
¥
    Notes¥
    -----¥
        If you want to approximate a mixture point of two vertices, you can specify the other point in target_pref_2.¥
        In this case, you should also indicate the mixture weight.¥
    """¥
    # set the target preference¥
    self.rho_set = rho_set¥
    self.rho = dict(sum([[((D, x), prob) for x, prob in zip(D, probs)] for D, probs in rho_set.items()], []))¥
¥
    # initialization¥
    alpha = lambda n: 1 / (n + 1)¥
¥
    self.z_list = [dict([(D, np.zeros(len(D))) for D in self.cal_D])]¥
    self.alpha_list = []¥
    self.beta_list = []¥
    self.rho_list = []¥
    self.l2_error_list = [np.inf]¥
¥
    for n in range(iter_num):¥
        # z^¥{n-1¥}¥
        rho_est_set_prev = self.z_list[-1]¥
        rho_est_prev = dict(sum([[((D, x), prob) for x, prob in zip(D, rho_est_set_prev[D])] for D in self.cal_D], []))
¥
¥
        # set target¥
        target_rho_set = dict([(D, (self.rho_set[D] - (1 - alpha(n)) * rho_est_set_prev[D]) / alpha(n)) for D in
self.cal_D])¥
¥
        # compute beta of z_n¥
        beta = self.update_beta(target_rho_set, fixed_effects)¥
¥
        # z_n¥
        rho_est_curr = self.get_rho_from_logit(beta, fixed_effects)¥
        rho_est_set_curr = dict([(D, np.array([rho_est_curr[(D, x)] for x in D])) for D in self.cal_D])¥
¥
        # z^n¥
        rho_set_est = dict([(D, (1 - alpha(n)) * rho_est_set_prev[D] + alpha(n) * rho_est_set_curr[D]) for D in
self.cal_D])¥
        rho_est = dict(sum([[((D, x), prob) for x, prob in zip(D, rho_set_est[D])] for D in self.cal_D], []))¥
¥
        # progress just for log¥
        error = self.l2_distance(self.rho, rho_est)¥
¥
        # store parameter records¥
        self.z_list.append(rho_set_est)¥
        self.alpha_list.append(alpha(n))¥
        self.beta_list.append(beta)¥
        self.l2_error_list.append(error)¥
```

```
¥
        # calculate lambdas¥
        self.lambda_list = []¥
        for n in range(iter_num):¥
            lam = self.alpha_list[n] * np.prod([1 - alpha for alpha in self.alpha_list[n+1:]])¥
            self.lambda_list.append(lam)¥
¥
    def update_beta(self, target_rho_set, fixed_effects=0):¥
        """¥
        Parameters¥
        ----------¥
            target_rho_set : dict¥
                a dictionary of which keys are choice sets¥
            fixed_effects : len(X)-dim array, default 0¥
                fixed effects¥
¥
        Returns¥
        -------¥
            beta_new : K-dim array¥
        """¥
        if not isinstance(fixed_effects, Iterable):¥
            fixed_effects = np.zeros(len(self.X))¥
¥
        def eq(beta):¥
            return sum([np.square(target_rho_set[D] - self.logit(D, beta, fixed_effects[[D]])).sum() for D in self.cal_D])¥
¥
        beta_init = np.random.normal(scale=0.1, size=self.K)¥
        method = "BFGS"¥
        res = minimize(eq, beta_init, bounds=[(-20, 20) for _ in range(self.K)], method=method)¥
        beta_new = res.x¥
        return beta_new¥
¥
¥
def process_EM(pref, features):¥
    """¥
    get the l2 error based on an EM algorithm¥
¥
    Parameters¥
    ----------¥
        pref : tuple¥
            a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
        features : array¥
            features of each alternative¥
¥
    Returns¥
    -------¥
        error : float¥
            the estimated l2 error¥
    """¥
    inst = ApproxRUM(features)¥
    inst.EM_algorithm_from_pref(pref, 10, 1000)¥
    beta_est_mat = inst.beta_mat_list[-1]¥
    lambda_est = inst.lambda_list[-1]¥
    rho_est = inst.get_rho_from_mixed_logit(lambda_est, beta_est_mat)¥
    error = inst.l2_distance(inst.rho, rho_est)¥
    return pref, error¥
¥
```

```
¥
def process_greedy(pref, features, fixed_effects=0):¥
    """¥
    get the l2 error based on a greedy algorithm¥
¥
    Parameters¥
    ----------¥
        pref : tuple¥
            a tuple with length n like (0, 1, 3, 2), which means 0 > 1 > 3 > 2¥
        features : array¥
            features of each alternative¥
        fixed_effects : len(X)-dim array, default 0¥
            fixed effects¥
¥
    Returns¥
    -------¥
        error : float¥
            the estimated l2 error¥
    """¥
    # pref = (2, 0, 1, 3)¥
    inst = ApproxRUM(features)¥
    inst.greedy_algorithm_from_pref(pref, 10, fixed_effects=fixed_effects)¥
    beta_est = inst.beta_list[-1]¥
    rho_est = inst.get_rho_from_logit(beta_est)¥
    error = inst.l2_distance(inst.rho, rho_est)¥
    return pref, error¥
¥
¥
# %%¥
if __name__ == "__main__":¥
    # Alternatives¥
    ranking_list = list(permutations((0, 1, 2, 3)))¥
¥
    # fish data¥
    features_original = np.array([¥
        [3995.915, 0.2410113,],¥
        [4044.080, 0.1712146,],¥
        [4014.958, 0.6293679,],¥
        [3995.915, 0.1622237,],¥
    ])¥
¥
    # d = 1¥
    features_1 = features_original¥
¥
    # d = 2¥
    features_2 = np.c_[¥
        features_original,¥
        features_original[:, 0] * features_original[:, 0], ¥
        features_original[:, 0] * features_original[:, 1], ¥
        features_original[:, 1] * features_original[:, 1],¥
    ]¥
¥
    #features = features_1  # mixed logit with degree 1¥
    features = features_2  # mixed logit with degree 2¥
¥
    # Outcomes depend on the initialization of parameters¥
    # Compute error for each of the rankings many times to avoid failure of optimization that is caused by bad initial
```

```
points¥
    num_init_choice = 5¥
    repeated_ranking_list = ranking_list * num_init_choice¥
¥
    # fixed_effects¥
    fixed_effects = np.array([0, 0, 0, 0])¥
¥
    # greedy¥
    # approximate all preferences¥
    # all rankings are representable for d = 2 (K = 2, |X| = 4)¥
    # errors for all vertices should be zero¥
    # but some of them are nonzero¥
    res_greedy = Parallel(n_jobs=-1, verbose=51)([delayed(process_greedy)(pref, features, fixed_effects) for pref in
repeated_ranking_list])¥
    df_error_greedy = pd.DataFrame(res_greedy).groupby(0).min()¥
   ¥
   # greedy algorithm seems not working for (0, 2, 3, 1), for example¥
   # error does not change even after you change number of iteration, M = 10, 100, 1000¥
   # order of error should be O(1/M) according to Haoge's note, right?¥
    pref, M = (0, 2, 3, 1,), 10¥
    inst = ApproxRUM(features)¥
    inst.greedy_algorithm_from_pref(pref, M, fixed_effects=fixed_effects)¥
    beta_est_mat, lambda_est = np.array(inst.beta_list), np.array(inst.lambda_list)¥
    rho_est = inst.get_rho_from_mixed_logit(lambda_est, beta_est_mat, fixed_effects)¥
    error = inst.l2_distance(inst.rho, rho_est)¥
}
```