

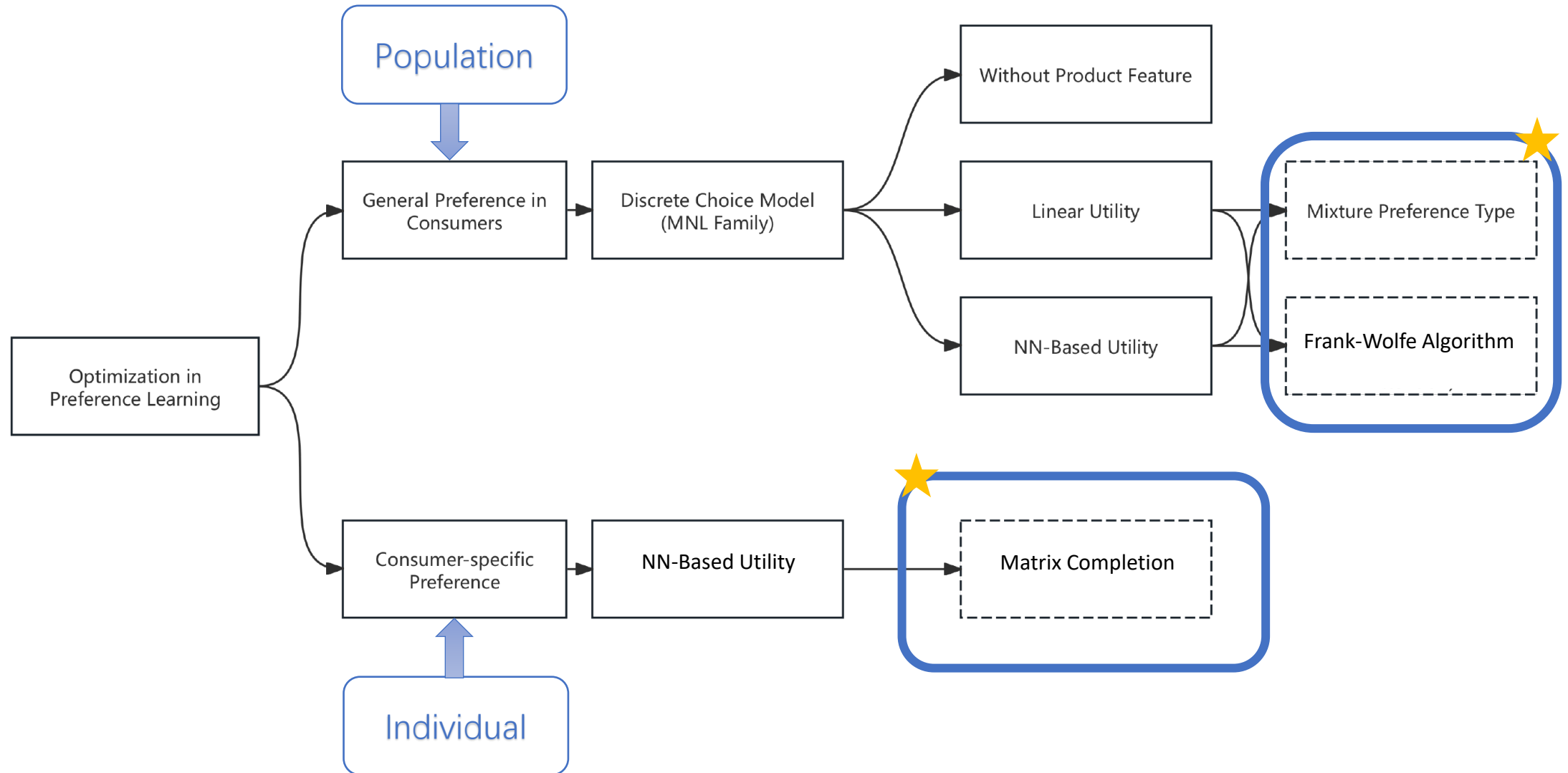
Choice Model and Preference learning

Shuhan Zhang zhang19@sas.upenn.edu

Xinyu Zhang joyxyz@sas.upenn.edu

Lexuan Chen amy0305@sas.upenn.edu

Outline



Preference Learning

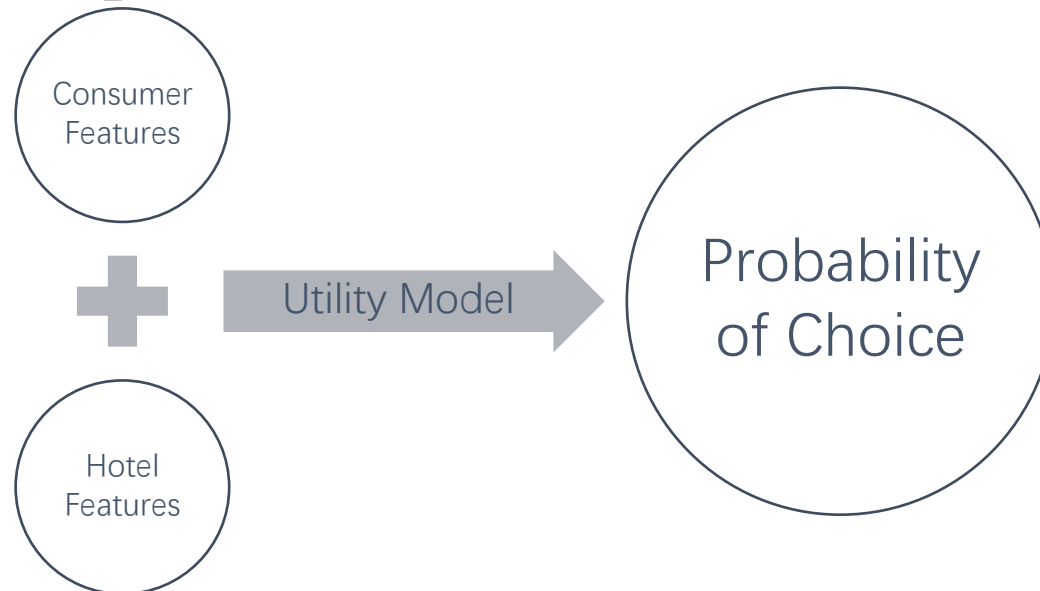
- Preference learning : to model and predict an individual's preferences or choices from a set of alternatives based on available data.
- Two types: population-wise & individual-wise
- Lots of applications.

What product the customers prefer and probably choose? → Preference Learning

Recommendation System, RLHF, Business Analytics, Assortment Optimization...

Problem Objective: What Problem Are We Solving?

- Predict consumer hotel preference from personalized offer sets
- Estimate **perceived utility** of each hotel based on features
- Calculate choice probabilities
- Minimize error between predicted and actual selection outcomes



An Example for Choice Model: Fruit Selection

Fruit (Sugar level)	Consumer	Offer Set	Choice
Apple (4)	A (diabetes)	Apple, Banana and Pear	?
Banana (5)	B (no diabetes)	Apple and Banana	?
Pear (3)			

- Objective: predict the consumer's choice
- Simple Example
- The Complexity: Scale up to thousands of products with various features, consumers with a range of personal features and much larger offer sets,
- Predicting the choice becomes much more challenging!

Why Is This Problem Important?

- Online platforms like Expedia present thousands of options — smart filtering is crucial
- It translates abstract consumer behavior into a quantifiable, interpretable optimization model grounded in economic theory.
- Real-world constraints: **limited user visibility, heterogeneous preferences, and missing data**



Experiments: Expedia Dataset

Feature Type	Feature Name	Variable Type	Description
Hotel Feature	PageRank	Discrete	The recommendation position of the hotel
	Star Rating	Continuous	The historical star rating
	Location Score	Continuous	The desirability of a hotel's location
	Historical Price	Continuous	Mean price of the hotel over the last period
	Branded	Binary	Whether the hotel belongs to a major hotel chain
	Promotion	Binary	Whether the hotel has a promotional price
	Price	Continuous	The displayed price of the hotel
	Booking	Binary	Whether the customer booked the hotel
Search Criterion	Booking Window	Discrete	Number of days in the future from the search date
	Length of Stay	Discrete	Number of nights of stay
	Adults Count	Discrete	Number of adults in the party
	Children Count	Discrete	Number of children in the party
	Room Count	Discrete	Number of rooms requested
	Saturday Night	Binary	Whether the stay includes a Saturday night
	Random	Binary	Whether the hotel list is sorted randomly

Table 1: Description of features used in the Expedia Hotel dataset.

General Setup

- N : the set of all products that a seller can offer to customers
- S : the set of products that the seller actually offers to the customer (offer set), $S \subseteq N$
- s : a specific product in offer set S , $s \in S$
- $P(s|S)$: The choice probability for product s given offer set S

Random Utility Theorem and MNL Model

- RUT (Random Utility Theorem)

Customers map a utility value u^s for each product s , and select the most valuable product in the given offer set.

- SoftMax function: map the utility value to probability space

$$P(s|S) = \frac{\exp(u^s)}{\sum_{s' \in S} \exp(u^{s'})}$$

Optimization Objective

- Minimize the Negative Log Likelihood (NLL)

$$\text{NLL}(g) = -\frac{1}{N} \sum_{t=1}^T \sum_{j \in S_t} N_{jt} \log g_{jt}$$

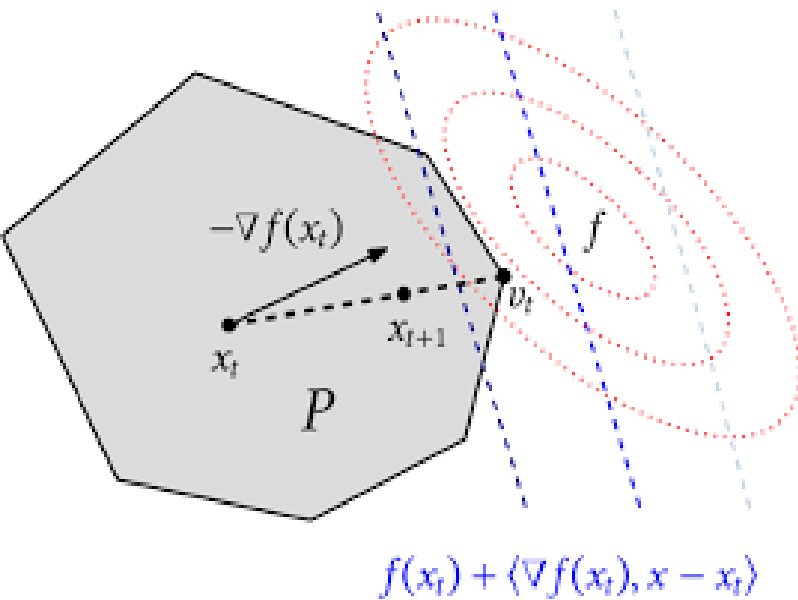
- N : Total number of product choices
- T : Total number of offer set
- g_{jt} : the predicted probability that product j is chosen in offer set t .
- N_{jt} : one-hot choice result vector
- The objective function is the negative log likelihood loss, which is commonly seen in classification model and preference learning tasks.

Model Candidate

1. $u^s = C^s$, **Constant utility**, scenario without feature & small N
2. $u^s = f(x^s)$, **Linear Function** $u^s = w^\top x^s$, **Neural Network**, scenario with feature x^s for product s , w is the weight matrix
3. **Mixture Preference Types**, α_k and u_k^s , $\sum_{k=1}^K \alpha_k = 1$, α_k is the proportion of preference type k in population
4. *Matrix Completion, preference inference among individuals based on NN results*

Mixture Preference Learning with *Frank-Wolfe* Algorithm

Shuhan Zhang



Outline

- Existing literature drawbacks
- Problem and optimization objective
- Algorithm framework & support finding
- Experiment setup
- Results
- Discussion

Reference

- **The idea are inspired by paper: *A Conditional Gradient Approach for Nonparametric Estimation of Mixing Distributions***
- We want to explore the effect of replacing linear utility function with NN-based function. Will it improve the performance?

A Conditional Gradient Approach for Nonparametric Estimation of Mixing Distributions

Management Science

Srikanth Jagabathula,^{a,b} Lakshminarayanan Subramanian,^c Ashwin Venkataraman^{c,d}

^aStern School of Business, New York University, New York, New York 10012; ^bHarvard Business School, Harvard University, Boston, Massachusetts 02163; ^cCourant Institute of Mathematical Sciences, New York University, New York, New York 10012; ^dHarvard Institute for Quantitative Social Science, Harvard University, Cambridge, Massachusetts 02138

Contact: sjagabat@stern.nyu.edu,  <http://orcid.org/0000-0002-4854-3181> (SJ); lakshmi@cs.nyu.edu (LS); ashwin@cs.nyu.edu (AV)

Why do We Need Frank-Wolfe Algorithm?

- Avoid Parametric Bias → No assumption of specific distribution
 - Specific distribution assumption introduces bias.
 - This can lead to inaccurate model predictions, especially when using methods like Expectation-Maximization (EM).
- Not being able to clearly formulate the problem's constraints.
 - For example, mapping weights and features to a probability space is complex and hard to describe explicitly.
 - Therefore, we can not use Projected Gradient Method.

Optimization Objective

- Negative Log-Likelihood (NLL) Loss

$$\text{NLL}(g) = -\frac{1}{N} \sum_{t=1}^T \sum_{j \in S_t} N_{jt} \log g_{jt}$$

Then the choice probability for product j in offer set S_t is given by:

$$g_{jt} = \sum_{k=1}^K \alpha_k \cdot f_{jt}(\omega_k)$$

where

$$f_{jt}(\omega_k) = \frac{\exp(\omega_k^\top z_{jt})}{\sum_{\ell \in S_t} \exp(\omega_k^\top z_{\ell t})}$$

Therefore,

$$g_{jt} = \sum_{k=1}^K \alpha_k \cdot \frac{\exp(\omega_k^\top z_{jt})}{\sum_{\ell \in S_t} \exp(\omega_k^\top z_{\ell t})}$$

- N : Total number of product choices
- T : Total number of offer set
- g_{jt} : the predicted probability that product j is chosen in offer set t .
- N_{jt} : one-hot choice result vector
- α_k : the proportion of preference type k in population
- ω_k : the weight matrix of features for preference type k
- z_{jt} : features of product j in offer set t

Algorithm framework

Algorithm 1 Conditional Gradient Algorithm for Estimating the Mixing Distribution

- 1: **Input:** Loss function $\text{loss}(g)$, initial guess $g^{(0)} \in \mathcal{F}$, max iteration K
- 2: **Initialize:** $k \leftarrow 0$, set $\alpha^{(0)} = 1$, mixing support $\mathcal{S} = \{g^{(0)}\}$
- 3: **while** stopping condition not met **do**
- 4: $k \leftarrow k + 1$
- 5: Compute gradient $\nabla \text{loss}(g^{(k-1)})$
- 6: Solve support-finding step:

$$f^{(k)} \leftarrow \arg \min_{f \in \mathcal{F}} \langle \nabla \text{loss}(g^{(k-1)}), f - g^{(k-1)} \rangle$$

- 7: Update support set: $\mathcal{S} \leftarrow \mathcal{S} \cup \{f^{(k)}\}$
- 8: Fully corrective update:

$$\alpha^{(k)} \leftarrow \arg \min_{\alpha \in \Delta^k} \text{loss} \left(\sum_{s=0}^k \alpha_s f^{(s)} \right)$$

- 9: Update solution:

$$g^{(k)} \leftarrow \sum_{s=0}^k \alpha_s^{(k)} f^{(s)}$$

- 10: **end while**
 - 11: **Output:** Mixing components $\{f^{(s)}\}_{s=0}^k$ and weights $\alpha^{(k)}$
-

Finding the direction the mixture distribution should develop



$$f^{(k)} \leftarrow \arg \min_{f \in \mathcal{F}} \langle \nabla \text{loss}(g^{(k-1)}), f - g^{(k-1)} \rangle$$



$$\alpha^{(k)} \leftarrow \arg \min_{\alpha \in \Delta^k} \text{loss} \left(\sum_{s=0}^k \alpha_s f^{(s)} \right)$$



Refit the proportion of each mixture component

?



$$\min_{\omega \in \mathbb{R}^d} \langle \nabla \text{loss}(g^{(k-1)}), f(\omega) \rangle$$



$$g^{(k)} \leftarrow \sum_{s=0}^k \alpha_s^{(k)} f^{(s)}$$

Support Finding = Finding new preference type

$$\boxed{\min_{f \in \text{conv}(\mathcal{F})} \langle \nabla \text{loss}(g^{(k-1)}), f \rangle} \quad \longrightarrow \quad \min_{f \in \mathcal{F}} \langle \nabla \text{loss}(g^{(k-1)}), f \rangle$$

- Linear minimization problem over the convex hull $\text{conv}(\mathcal{F})$ of atomic likelihood vectors
- Optimum always achieved at one of its extreme points!

$$\mathcal{F} := \{f(\omega) \mid \omega \in \mathbb{R}^d\}$$

- Extreme lies in \mathcal{F} .

One Step Further: NN-based Support Finding

- We use $f_{\theta}(x) = \text{softmax}(W_2 \cdot \sigma(W_1 x + b_1) + b_2)$ to approximate

$$f_{jt}(\omega_k) = \frac{\exp(\omega_k^{\top} z_{jt})}{\sum_{\ell \in S_t} \exp(\omega_k^{\top} z_{\ell t})}$$

- $\theta = \{W_1, W_2, b_1, b_2\}$ are the learnable parameters of the network.
- $\sigma(\cdot)$ is a nonlinear activation function (e.g., ReLU or tanh).
- Instead of solving for the optimal logit parameter ω , we optimize the neural network parameters θ to minimize the same linear objective:

$$\min_{\theta} \left\langle \nabla \text{loss}(g^{(k-1)}), f_{\theta}(x) \right\rangle$$

Experiments: Expedia Dataset

Feature Type	Feature Name	Variable Type	Description
Hotel Feature	PageRank	Discrete	The recommendation position of the hotel
	Star Rating	Continuous	The historical star rating
	Location Score	Continuous	The desirability of a hotel's location
	Historical Price	Continuous	Mean price of the hotel over the last period
	Branded	Binary	Whether the hotel belongs to a major hotel chain
	Promotion	Binary	Whether the hotel has a promotional price
	Price	Continuous	The displayed price of the hotel
	Booking	Binary	Whether the customer booked the hotel
Search Criterion	Booking Window	Discrete	Number of days in the future from the search date
	Length of Stay	Discrete	Number of nights of stay
	Adults Count	Discrete	Number of adults in the party
	Children Count	Discrete	Number of children in the party
	Room Count	Discrete	Number of rooms requested
	Saturday Night	Binary	Whether the stay includes a Saturday night
	Random	Binary	Whether the hotel list is sorted randomly

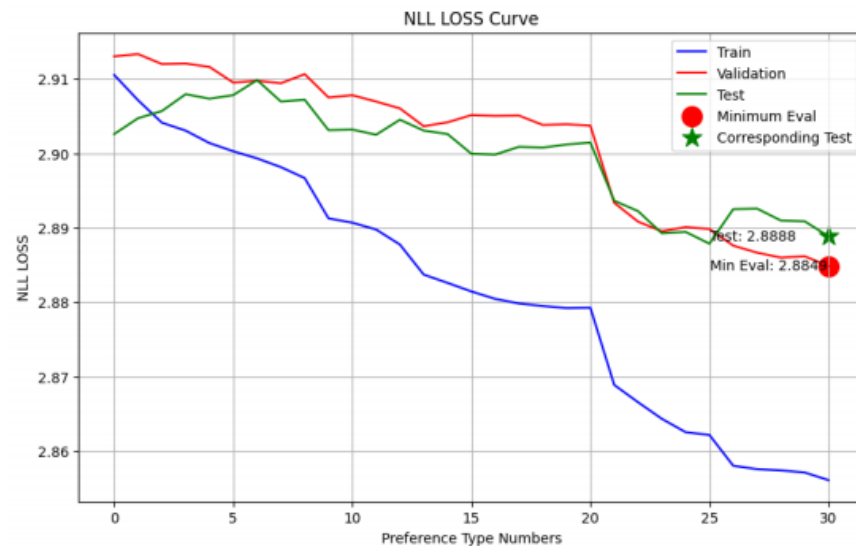
Table 1: Description of features used in the Expedia Hotel dataset.

General Settings

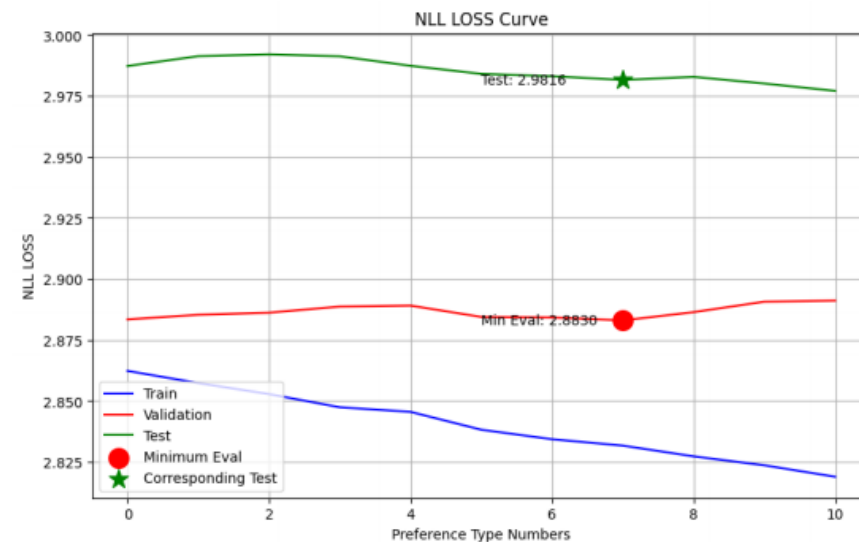
Component	Configuration
Dataset Split	8000 training / 2000 validation / 1000 testing
Baseline Model	Linear logit model with 30 types
NN Model Types	2-layer feedforward networks
Hidden Sizes Tested	{30, 10, 5}
Activation Function	ReLU
Atomic Likelihood Function	Softmax over network output
Support Finding Optimizer	L-BFGS (full-batch)
Proportion Update Optimizer	Adam (full-batch)
Initialization	Adam (mini-batch training)
Model Selection	Best validation performance after each type added
Final Evaluation	Best model tested on held-out test set

Table 2: Summary of experimental setup and training configurations.

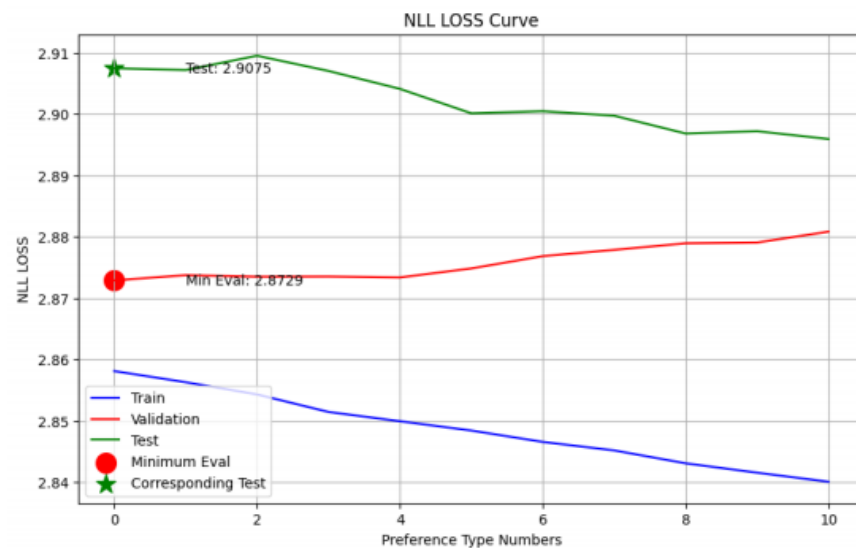
Results



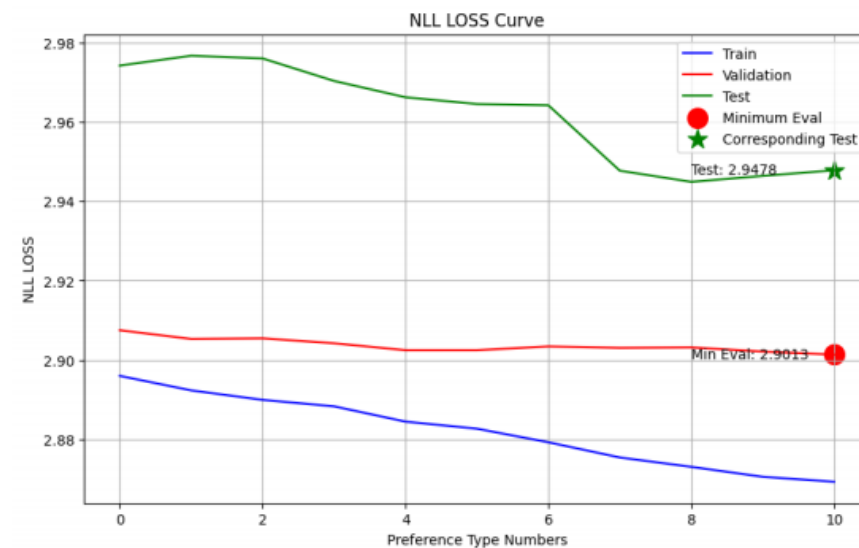
(a) Linear MNL Model (30 types)



(b) NN-based ($h=30$), 10 types



(c) NN-based ($h=10$), 10 types



(d) NN-based ($h=5$), 10 types

Figure 2: Comparison of Frank-Wolfe Mixture Models under different architectures.

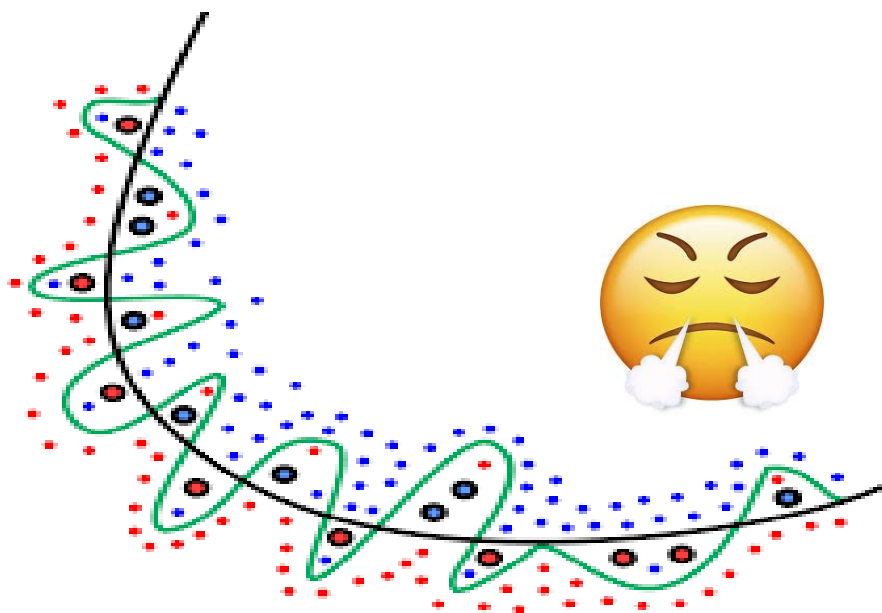
Results: Linear Mixture Model Wins

Model	Train	Validation	Test
Linear	2.9119	2.9133	2.8990
NN-30h	2.8616	2.8987	3.2475
NN-10h	2.8647	2.8860	3.1097
NN-5h	2.8641	2.8798	2.9629
Linear - 30 types	2.8561	2.8849	2.8888
NN-30h - 10 types	2.8190	2.8830	2.9816
NN-10h - 10 types	2.8400	2.8729	2.9075
NN-5h - 10 types	2.8692	2.9013	2.9478
Baseline - random guessing	3.5066	3.5066	3.5066

Table 3: Final performance values (NLL, validation, and test) for each model.

Discussion: Overfitting

Although the neural network-based support-finding model is more expressive than its linear counterpart, our experiments reveal that it is highly prone to overfitting—even with a small hidden size and substantial L_2 regularization. This observation holds consistently across all tested configurations.



Discussion: Analysis

- **Limited Data Size**

- Expedia: A limited number of training examples.
- Neural networks typically require large-scale data to generalize well.
- Poor generalization, even with strong regularization.

- **Under-regularized Proportions**

- We didn't penalize model complexity during the proportion update step.
- Overfitting: complex types receive large weights in the mixture even if they overfit the training data.
- Future work may include sparsity-inducing penalties or entropy regularization in the weight optimization process.

Discussion: Analysis

- **Inherent Linearity of Human Decision Boundaries**

- Human preferences in real-world decision-making (e.g., hotel booking) often follow relatively linear structures.
- Customers tend to prefer cheaper hotels, higher star ratings, or prominent locations.
- In such scenarios, linear models may already capture most of the signal, and introducing nonlinearity could add noise and increase the risk of overfitting.

Code Showing

```
class Problem_FrankWolfe:
    def __init__(self, p_offerset, p_sell, p_mask):
        # Define a dataset and sales data
        self.sales = Sales(p_offerset, p_sell, p_mask)
        self.dataset = CustomDataset(self.sales.offerset, self.sales.N_sales, self.sales.mask)
        self.train_loader = DataLoader(self.dataset, batch_size=1024, shuffle=True)

        # Define the main problem NLL loss
        self.NLL_main = None

        # Define the current likelihood convex combination
        self.g = None

        # Define the current likelihood gradient for support finding
        self.NLL_gradient = None

        # Define a list to contain all choice likelihood
        self.fw_list = []

        # Define a list to contain all taste vector
        self.taste_list = []

        # Define a list to contain all proportion
        self.proportion = []
```

```
def init(self, val_offerset, val_sell, val_mask, patience=9):
    # We initialize with training a single NN Model
    initial_preference = Preference(self.sales.num_features)

    print('Initial Training Begin')
    criterion = nn.NLLLoss()
    optimizer = optim.Adam(initial_preference.parameters(), lr=1e-2)
    num_epochs = 200

    best_val_loss = float('inf') # Initialize best validation loss
    early_stop_counter = 0 # Initialize early stopping counter
    val_loss_list = [] # List to store validation loss values
    train_loss_list = [] # List to store train loss values
    for epoch in range(num_epochs):
        for batch_idx, (offerset, sell, mask) in enumerate(self.train_loader):
            log_choice_p = initial_preference(offerset, mask)
            sell = sell.type(torch.int64).argmax(dim=1)
            loss = criterion(log_choice_p, sell)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Evaluate on validation set
        with torch.no_grad():
            log_choice_p_val = initial_preference(torch.tensor(val_offerset, dtype=torch.float32, device=device), torch.tensor(val_mask, dtype=torch.float32, device=device))
            val_loss = criterion(log_choice_p_val, torch.tensor(val_sell, dtype=torch.int64, device=device).argmax(dim=1))
            val_loss_list.append(val_loss.item()) # Append validation loss to the list

            log_choice_p_train = initial_preference(self.sales.offerset, self.sales.mask)
            train_loss = criterion(log_choice_p_train, self.sales.N_sales.type(torch.int64).argmax(dim=1))
            train_loss_list.append(train_loss.item()) # Append train loss to the list
```

```
log_choice_p_train = initial_preference(self.sales.offerset, self.sales.mask)
train_loss = criterion(log_choice_p_train, self.sales.N_sales.type(torch.int64).argmax(dim=1))
train_loss_list.append(train_loss.item()) # Append train loss to the list

# Check for early stopping
if val_loss < best_val_loss:
    best_val_loss = val_loss
    early_stop_counter = 0
else:
    early_stop_counter += 1
    if early_stop_counter >= patience:
        print(f'Early stopping at epoch {epoch + 1}')
        break

if (epoch + 1) % 10 == 0:
    print(f'Epoch {epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}, Val Loss: {val_loss.item():.4f}')

print('Initial Training End')

# Add the initial consumer type in the consumer list
self.taste_list.append(initial_preference)
self.fw_list.append(torch.exp(initial_preference(self.sales.offerset, self.sales.mask))) # Use initial_preference here
```

```
def main_problem_loss(self):
    N_sales = self.sales.N_sales
    with torch.no_grad():
        f = torch.zeros(N_sales.shape, dtype=torch.float32, device=device)
        for proportion, fw in zip(self.proportion, self.fw_list):
            f += proportion * fw
        f.requires_grad = True
        f.log = torch.log(f)
        self.NLL_main = nn.NLLLoss()(f.log, N_sales.type(torch.int64).argmax(dim=1))
        self.NLL_main.backward()
        self.NLL_gradient = f.grad.clone()
        self.NLL_gradient.requires_grad = False
        self.NLL_gradient = torch.where(torch.isnan(self.NLL_gradient), torch.tensor(0.0), self.NLL_gradient)
    # update the dataset for support finding
    self.dataset = CustomDataset(self.sales.offerset, self.NLL_gradient, self.sales.mask)
    self.train_loader = DataLoader(self.dataset, batch_size=1024, shuffle=True)
    return self.NLL_main

def support_finding_loss(self, p_choice_p, p_gradient):
    pre_loss = p_gradient * p_choice_p
    pre_loss = torch.where(torch.isnan(pre_loss), torch.tensor(0.0), pre_loss)
    return torch.sum(pre_loss)

def proportion_update_loss(self, alpha):
    alpha = F.softmax(alpha, dim=0)
    N_sales = self.sales.N_sales
    fw_tensor = torch.stack(self.fw_list, dim=0)
    chosen_indices = N_sales.type(torch.int64).argmax(dim=1)
    chosen_probs = fw_tensor[:, torch.arange(fw_tensor.shape[1]), chosen_indices]
    mixed_probs = torch.sum(chosen_probs * alpha, dim=0)
    log_probs = torch.log(mixed_probs * 1e-10)
    loss = -torch.sum(log_probs) / self.sales.num_offers
    return loss

def support_finding_lbfgs(self, reg_lambda=0):
    print('-----Consumer Type Search Begin-----')
    new_preference = Preference(self.sales.num_features)
    criterion = self.support_finding_loss
    optimizer = optim.LBFGS([new_preference.parameters()], lr=0.1, max_iter=20, history_size=100, line_search_fn='strong_wolfe') # Using LBFGS
    choice_p = None

    def closure():
        optimizer.zero_grad()
        log_choice_p = new_preference(self.sales.offerset, self.sales.mask)
        choice_p = torch.exp(log_choice_p)
        loss = criterion(choice_p, self.NLL_gradient)
        l2_reg = 0.0
        for param in new_preference.parameters():
            l2_reg += torch.norm(param, 2) # Calculate L2 norm
        loss += reg_lambda * l2_reg # Add regularization term to loss
        loss.backward()
        return loss

    num_epochs = 20 # You can adjust this as needed
    for epoch in range(num_epochs):
        loss = optimizer.step(closure)
        if (epoch + 1) % 1 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')
        with torch.no_grad():
            self.taste_list.append(new_preference)
            self.fw_list.append(torch.exp(new_preference(self.sales.offerset, self.sales.mask)))
        print('-----Consumer Type Search End-----')

def proportion_update(self):
    print('-----Proportion Update Search Begin-----')
    # =====
    alpha = torch.empty((len(self.taste_list), 1), dtype=torch.float32, requires_grad=True)
    init.uniform_(alpha, 0, 0)
    optimizer = optim.Adam([alpha], lr=5e-2)

    epoches = 500
    for epoch in range(epoches):
        loss = self.proportion_update_loss(alpha)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (epoch + 1) % 100 == 0:
            print(f'Epoch {(epoch + 1)/epoches}], Loss: {loss.item():.4f}')
            self.proportion = F.softmax(alpha, dim=0).flatten().tolist()
            print(self.proportion)
            self.main_problem_loss()
            print('-----Proportion Update Search End-----')
    # =====
```

```
# define the sales class to store the information of the dataset
class Sales:
    def __init__(self, p_offerset, p_sell, p_mask):
        # N_sales (num_offers, num_products)
        self.N_sales = torch.tensor(p_sell, dtype=torch.float32)
        # offerset (num_offers, num_products, num_features)
        self.offerset = torch.tensor(p_offerset, dtype=torch.float32)
        # mask (num_offers, num_products)
        self.mask = torch.tensor(p_mask, dtype=torch.float32)

        # Some key information
        self.num_offers = self.offerset.shape[0]
        self.num_products = self.offerset.shape[1]
        self.num_features = self.offerset.shape[2]
        self.N = torch.sum(self.N_sales)
```

```
class Preference(nn.Module):
    def __init__(self, p_num_feature):
        super(Preference, self).__init__()
        self.linear = nn.Linear(p_num_feature, 1)
        init.normal_(self.linear.weight, std=1)

    def forward(self, p_offerset, p_mask):
        output = self.linear(p_offerset).squeeze(-1)
        masked_e = torch.where(p_mask == 1, output, float('-inf'))
        log_choice_p = F.log_softmax(masked_e, dim=-1)
        return log_choice_p
```

```
def get_offer_data(data_para):
    offerset_list = []
    sell_list = []
    mask_list = []
    max_num = 32
    for srch_id, group in data_para:
        num_product = len(group)
        # parallel offerset size
        offerset = group.drop(columns=['booking_bool', 'srch_id']).values
        offer_dummy = np.zeros((max_num - num_product, offerset.shape[1]))
        offerset = np.vstack((offerset, offer_dummy))
        offer_valid_mask = np.append(np.ones(num_product), np.zeros(max_num - num_product))

        # parallel offerset market share
        num_sell = group['booking_bool'].values
        num_sell_dummy = np.zeros((max_num - num_product))
        num_sell = np.hstack((num_sell, num_sell_dummy))

        offerset_list.append(offerset)
        sell_list.append(num_sell)
        mask_list.append(offer_valid_mask)

    offerset_list = np.array(offerset_list)
    mask_list = np.array(mask_list)
    sell_list = np.array(sell_list)

    return offerset_list, sell_list, mask_list
```

```
class CustomDataset(Dataset):
    def __init__(self, offerset_tensor, sell_tensor, mask_tensor):
        self.offerset_tensor = offerset_tensor
        self.sell_tensor = sell_tensor
        self.mask_tensor = mask_tensor
```

Preference Learning













A New Trial --- Matrix Completion

Lexuan Chen

Matrix Completion







Rating Matrix

X
 $n \times m$

						
	4	3		?	5	
	5		4		4	
	4		5	3	4	
		3				5
		4				4
			2	4		5







User Matrix

U
 $n \times k$

Item Matrix

V^T
 $k \times m$

Modeling assumption

$$X \approx UV^T$$

$$X \in \mathbb{R}^{n \times m}$$

$$U \in \mathbb{R}^{n \times k}$$

$$V \in \mathbb{R}^{m \times k}$$

k : rank of X

First matrix ---- each row belongs to an individual

Second matrix ---- each column belongs to an item

Netflix Dataset
(Film Puzzle)

Complete U and V \longrightarrow Predict completed X

MNL with 3 layer NN-based Model

We use a **3-layer feedforward neural network** to model the utility score of each hotel given a consumer's search offer set.

**Utility Score
Generation**

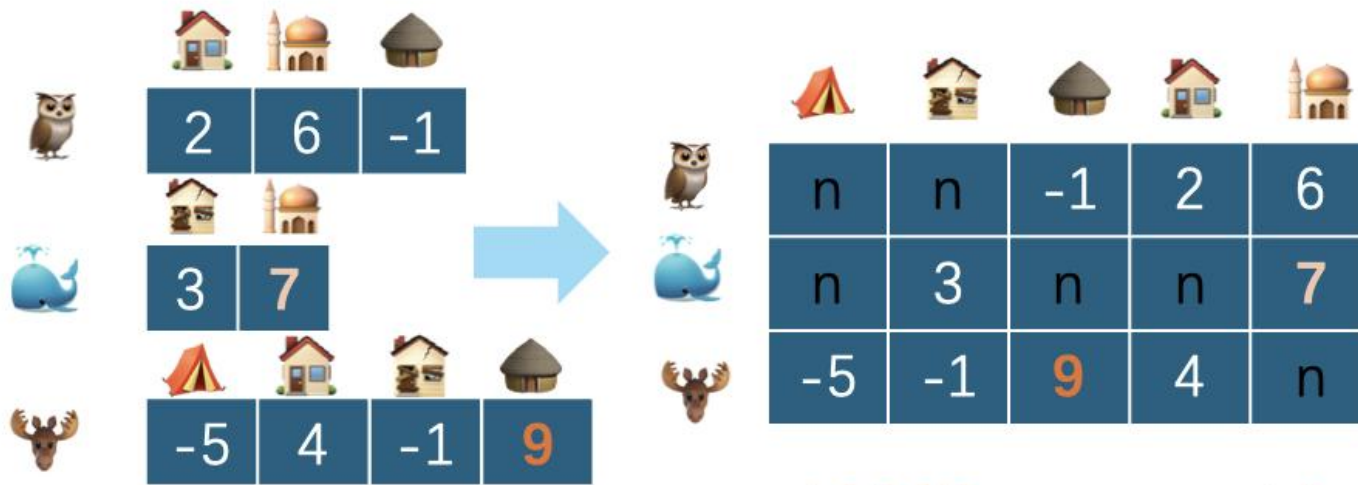
Input Features

Joint feature vector of a (*consumer, hotel*) pair

- **Consumer-side:** srch_length_of_stay, srch_booking_window, etc.
- **Hotel-side:** price_usd, location_score, promotion etc.

$$f(x) = W_3 \cdot \sigma(W_2 \cdot \sigma(W_1 x))$$

Layer	Description
Layer 1	Linear: $\mathbb{R}^d \rightarrow \mathbb{R}^{100}$ followed by Sigmoid
Layer 2	Linear: $\mathbb{R}^{100} \rightarrow \mathbb{R}^{100}$ with Sigmoid
Output	Linear: $\mathbb{R}^d \rightarrow \mathbb{R}^1$, outputs utility score



“item bias”

“individual bias”

A highly sparse matrix

Expedia
Dataset
(Hotel Puzzle)

Consumers’
Utility over Their
Own Offer Sets

Matrix View of the
Utility from all
(consumer, hotel)
Pairs

Train/Test
Split

Original
Matrix

Goal

Use matrix completion techniques in Expedia:

- Recover missing preferences
- **Keep** recovered matrix **low-rank (k)**

$$X \approx UV^T$$

$$X \in \mathbb{R}^{n \times m}$$

$$U \in \mathbb{R}^{n \times k}$$

$$V \in \mathbb{R}^{m \times k}$$

k: latent factors

Optimization Objectives

Train/Test Split

- $X \in \mathbb{R}^{m \times n}$: Partially observed utility matrix
- P : Set of observed entries
- $T \subset P$: Held-out test entries
- $\Omega = P \setminus T$: Training entries used in loss

Test set is never seen during training and is used to evaluate generalization on real missing entries.

Two types loss functions

- Loss is computed only over $(i, j) \in \Omega$
 - Unobserved entries (i.e., NaNs) are excluded from loss

• MSE

$$\mathcal{L}(U, V) = \frac{1}{|\Omega|} \sum_{(i,j) \in \Omega} \left(X_{ij} - (\hat{X})_{ij} \right)^2 + \lambda (\|U\|_F^2 + \|V\|_F^2)$$

- X_{ij} : Observed entry in the matrix (from data)
- \hat{X}_{ij} : Predicted score from the model
- λ : Regularization coefficient controlling model complexity

Use both in synthetic data and real-world datasets

• Huber

$$\mathcal{L}(U, V) = \frac{1}{|\Omega|} \sum_{(i,j) \in \Omega} \text{Huber}(\hat{X}_{ij} - X_{ij}, \delta) + \lambda (\|U\|_F^2 + \|V\|_F^2)$$

$$\text{Huber}(\hat{X}_{ij} - X_{ij}, \delta) = \begin{cases} \frac{1}{2} (\hat{X}_{ij} - X_{ij})^2 & \text{if } |\hat{X}_{ij} - X_{ij}| \leq \delta \\ \delta \left(|\hat{X}_{ij} - X_{ij}| - \frac{1}{2} \delta \right) & \text{otherwise} \end{cases}$$

Used in sparse, noisy real-world settings for robustness.

Two-stage Experiment

Method List

- Singular Value Thresholding (SVT)
- Gradient Descent-based Factorization with Different update strategies

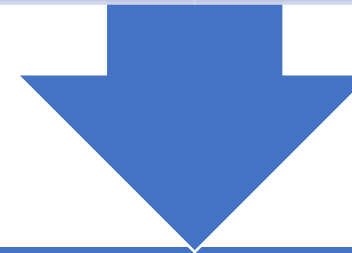
Baseline

Benchmark our best matrix factorization method against a popular SVT baseline on the Netflix dataset from Kaggle.

Synthetic and Benchmark Datasets

Try different methods in Synthetic Data

Use Netflix Dataset to Validate



Real world Expedia Dataset

Generate
Original
Score

Derive
Original
Matrix

Implement
Matrix
Completion

Stage 1: Synthetic and Benchmark Datasets

• Synthesis Method

Synthetic Data Generation. We generated synthetic data by multiplying two random matrices $U \in \mathbb{R}^{m \times r}$ and $V \in \mathbb{R}^{r \times n}$:

$$M = UV^\top \quad (2)$$

where $M \in \mathbb{R}^{m \times n}$ is the ground-truth low-rank matrix. We then introduced missingness by uniformly sampling a fraction ρ of the entries and replacing them with NaNs. The final observation matrix X used for recovery is:

$$X_{ij} = \begin{cases} M_{ij}, & \text{if } (i, j) \in \Omega \\ \text{NaN}, & \text{otherwise} \end{cases} \quad (3)$$

where Ω is the set of observed indices.

• First Try: Singular Value Thresholding (SVT)

A convex relaxation of the matrix completion problem

$$\min_X \lambda \|X\|_* + \frac{1}{2} \|X\|_F^2 \quad \text{s.t.} \quad \mathcal{P}_\Omega(X) = \mathcal{P}_\Omega(M)$$

- $\|X\|_*$: Nuclear norm (sum of singular values)
- $\|X\|_F$: Frobenius norm (square root of sum of squared entries)
- $\mathcal{P}_\Omega(\cdot)$: Projection operator that retains only observed entries in Ω

Traditional SVT methods are **slow** due to **iterative singular value thresholding**, and often yield **overly high-rank solutions** that compromise interpretability.

Stage 1: Synthetic and Benchmark Datasets

• Gradient Descent-based Factorization

Prediction formula $\hat{X}_{ij} = b + b_{u_i} + b_{v_j} + (UV^T)_{ij}$

Global bias $b = \frac{1}{|\Omega|} \sum_{(i,j) \in \Omega} X_{ij}$ User bias $b_{u_i} = \text{mean}(X_{i*}) - b$ Item bias $b_{v_j} = \text{mean}(X_{*j}) - b$

Different update strategies

Small Gaussian noise is added to better model the real world data

The following update strategies were compared:

- **GD:** Standard gradient descent.
- **Momentum:** Adds a velocity term to accelerate convergence:

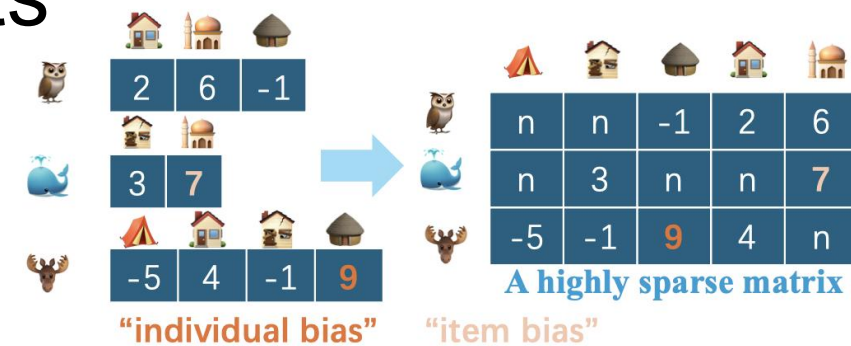
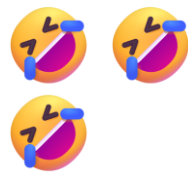
$$v_t = \beta v_{t-1} + \nabla L, \quad \theta \leftarrow \theta - \alpha v_t + \epsilon$$

- **EMA:** Applies exponential moving averages directly to the parameter updates:

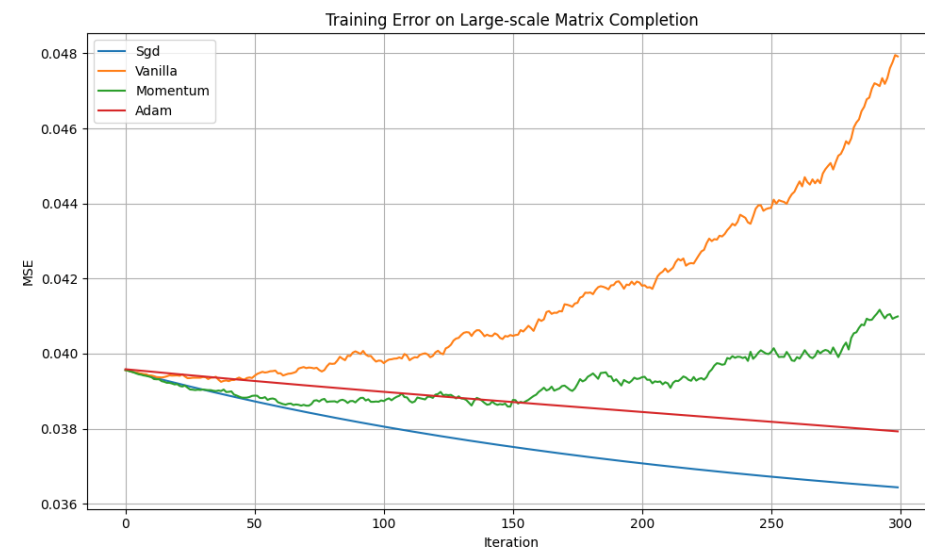
$$\theta_{\text{ema}} \leftarrow \theta_{\text{ema}} - \alpha \nabla L, \quad \theta \leftarrow (1 - \gamma) \theta_{\text{ema}} + \gamma \theta + \epsilon$$

- **Adam:** Combines momentum and adaptive learning rate (not manually derived but used for baseline comparison).

Winner: Adam optimizer



Bias-aware initialization to extract the influence of users' and items' individual characteristic



We adopt Adam for downstream real-world matrix completion tasks.

Stage 1: Netflix Validation

Netflix Grid Search

k	ℓ_2	Loss	LR	RMSE	Epoch
30	0.0010	MSE	0.001	0.8615	197
50	0.0001	Huber	0.001	0.8619	154
30	0.0005	MSE	0.001	0.8619	189
20	0.0001	MSE	0.001	0.8621	220
20	0.0005	Huber	0.001	0.8626	218

Table 1: Top-5 grid search results on Netflix.

Predictive Rankings

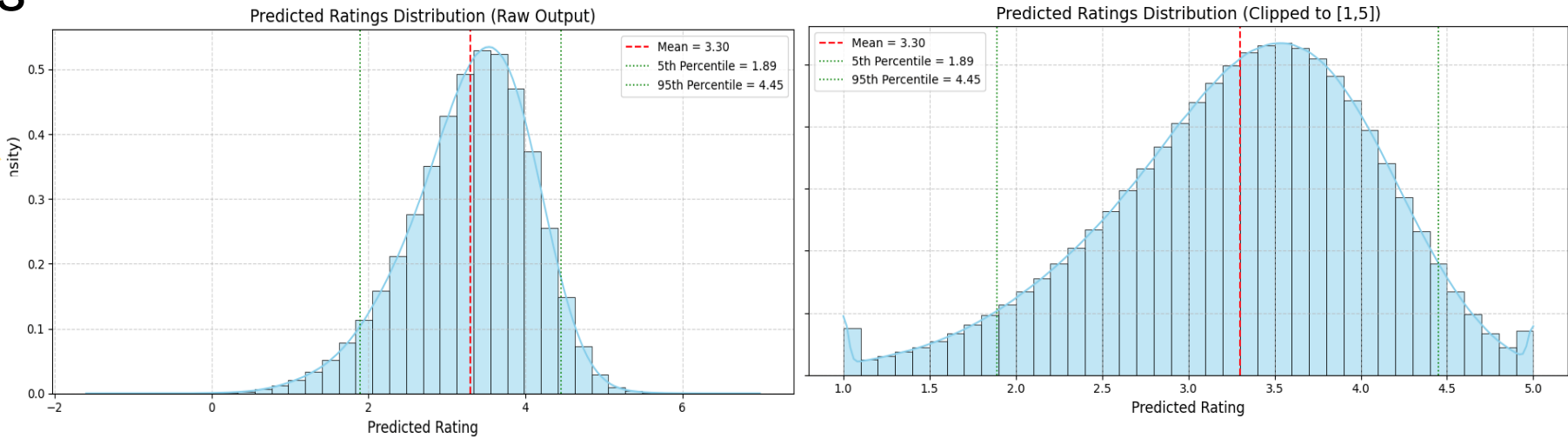
Rank	Title	Predicted Rating
1	Song of Freedom (1936)	5.17
2	Lured (1947)	5.07
3	Brother of Sleep (1995)	5.06
4	Gate of Heavenly Peace, The (1995)	5.03
5	Follow the Bitch (1998)	5.03
6	Mamma Roma (1962)	5.02
7	One Little Indian (1973)	4.99
8	Bittersweet Motel (2000)	4.90
9	Smashing Time (1967)	4.84
10	Baby, The (1973)	4.82

Table 2: Top-10 movie predictions from matrix completion on Netflix.

Netflix Benchmark: Adam vs SVT

- Baseline (SVT – Kaggle): RMSE = **2.1635** on 2,500 test entries
- Our Model (Adam MF): RMSE = **0.8644**
 - * $k = 30$, learning rate = 10^{-3}
 - * Loss: MSE, $\ell_2 = 10^{-3}$

Strong real-world performance → gave us confidence to apply the method to **Expedia**



Stage 2: Matrix Completion in Expedia

- Expedia Grid Search

k	ℓ_2	Loss	LR	RMSE	Epoch
30	0.0001	Huber	0.01	0.6109	27
30	0.0010	Huber	0.01	0.6132	26
20	0.0010	Huber	0.01	0.6145	30
30	0.0005	Huber	0.01	0.6149	27
20	0.0001	Huber	0.01	0.6158	31

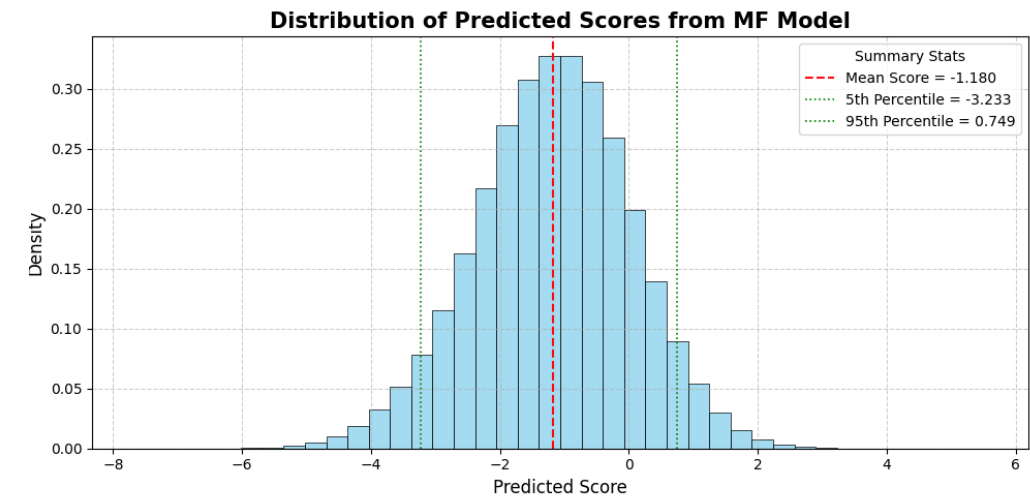
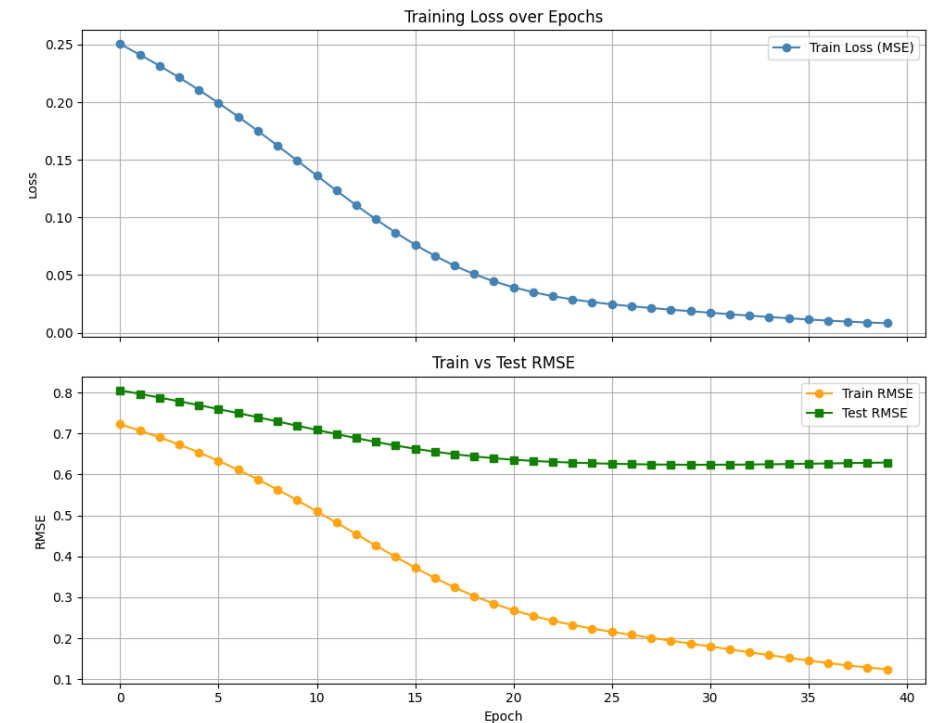
Table 3: Top-5 configurations using Huber loss on Expedia data.

Empirically, grid search results showed that Huber consistently outperformed MSE in terms of final RMSE

- Discussion on Early Stopping

Early stopping triggered at **epoch 27**.

- **Extreme sparsity** of the matrix
- **Noisy targets**
- Learned user/item biases may have absorbed most variation → Latent factors converged quickly



Stage 2: Matrix Completion in Expedia

- Overall Reflections

- Expedia's matrix is **extremely sparse** ($< 1\%$), below recovery thresholds
- Scores are **model-derived**, not ground-truth feedback
- **Clustering** users/items may improve coherence before completion

Even under sparsity and noise labels, matrix completion proved viable and interpretable.

References

- Y. Chi, "Low-rank matrix completion," IEEE Signal Processing Magazine, vol. 35, no. 5, pp. 178–181, 2018.
- J.-F. Cai, E. J. Candès, and Z. Shen, "A singular value thresholding algorithm for matrix completion," arXiv preprint arXiv:0810.3286, 2008.
- O. Golden, "Netflix Problem – Singular Value Thresholding," Kaggle Notebook, <https://www.kaggle.com/code/odedgolden/netflix-problem-singular-value-thresholding/notebook>

Self Critiques-Shuhan Zhang

- What was the most surprising result or finding during the project?
- The most surprising finding was that neural networks performed worse than linear models on the preference learning task in terms of generalization. Despite their expressive power, neural networks were highly prone to overfitting in this setting. This revealed a key insight: neural networks are not universally superior, and there is no free lunch in machine learning. The inherent linearity in human decision-making gave linear models a natural advantage in this context.
- Which specific lecture topic, concept, or technique from the course was most useful for your project? Explain how.
- The lecture content on optimizers was particularly useful. In my project, I experimented with both L-BFGS and Adam, as well as different optimization strategies including full-batch and mini-batch training. Each optimizer and strategy affected performance differently, and understanding their mechanics helped me better tune my models for the specific task.

Self Critiques-Shuhan Zhang

- How has your perspective on applying optimization in practice changed since the beginning of the course?
- At the beginning of the course, I saw optimization as a standalone algorithmic problem. However, I've come to realize that optimization is deeply tied to the nature of the model and the task at hand. There is no universally best optimizer—each comes with its own trade-offs in convergence speed, generalization, and computational efficiency. Choosing and tuning an optimizer is as much an art as it is a science.
- If you had two more weeks, what specific next step would you take on the project?
- Given more time, I would apply the learned choice model to an assortment optimization problem—using predicted preferences to select product combinations that maximize overall utility or profit. This is a natural and practical downstream application of our model.
- If you could restart the project, what is the single biggest change you would make to your approach or plan?
- If I could start over, I would switch away from the Expedia dataset earlier for the matrix completion task. It turned out to be a poor fit for this purpose, and relying on synthetic data was not very informative. A better real-world dataset could have led to more meaningful insights.

Self Critiques-Xinyu Zhang

- 1. What was the most surprising result or finding during the project?
- One of the most surprising findings was that the model will not improve greatly when increasing preference type after around 10 preference types while the optimal number of preference type is not fixed, showing that the data structure is different for the train-test split. Contrary to expectations, increasing the hidden layer size from 5 to 30 did not enhance predictive accuracy. The model with 30 hidden units overfit the training data and performed the worst on the test set.
- 2. Which specific lecture topic, concept, or technique from the course was most useful for your project? Explain how.
- The most impactful concepts were the practical treatment of batch size in stochastic optimization and the application of PyTorch for model construction, which allowed for flexible experimentation with neural network architectures and gradient-based methods throughout the project.
- 3. How has your perspective on applying optimization in practice changed since the beginning of the course?
- My initial view of optimization was algorithm-centric—focusing on mechanics like gradient descent or convexity. This course reshaped that view: I now understand that effective optimization is problem-specific, requiring alignment between the model structure, data characteristics, and the chosen optimizer. Selecting and tuning an optimizer is a process that balances computational efficiency, convergence behavior, and generalization.

Self Critiques-Xinyu Zhang

- 4. If you had two more weeks, what specific next step would you take on the project?
- With two additional weeks, I would explore alternative neural network architectures—such as residual connections or attention mechanisms—rather than simply scaling hidden units. I would also incorporate explicit regularization strategies (e.g., dropout or weight penalties) to mitigate overfitting and improve out-of-sample robustness.
- 5. If you could restart the project, what is the single biggest change you would make to your approach or plan?
- If I could restart, I would place greater emphasis on evaluating model generalization through cross-validation and ablation studies. Our early evaluations focused too heavily on final test accuracy, without systematically analyzing which components—model depth, type count, feature interaction—actually drove performance. A more principled experimental framework would have led to clearer insights and more reliable conclusions.

Self Critiques-Lexuan Chen

- 1. What was the most surprising result or finding during the project?
- Despite extreme sparsity in observed entries, matrix completion via gradient descent produced reliable reconstructions. To make this scalable, I introduced a mini-batch training structure—adapting deep learning techniques to the matrix setting—which enabled efficient GPU acceleration without compromising accuracy. Through grid search, I further identified Huber loss as more robust than MSE under real-world noise, reinforcing the method’s reliability on behavioral data.
- 2. Which specific lecture topic, concept, or technique from the course was most useful for your project? Explain how.
- PyTorch usage was critical—enabling all model training with GPU acceleration. Optimizer lectures helped me implement and analyze GD, Momentum, EMA, and Adam in matrix settings. Comparing their behavior under noise and scale deepened my understanding of practical optimization.
- 3. How has your perspective on applying optimization in practice changed since the beginning of the course?
- I used to view optimization as simply solving equations—an abstract mathematical procedure aimed at finding the best solution. Through this course, I’ve come to see it as a full modeling framework: defining objectives, designing loss functions, analyzing gradient flows, and diagnosing convergence. Optimization is no longer just a tool, but a mindset—a dynamic and creative process central to structuring, training, and refining quantitative models.

Self Critiques-Lexuan Chen

- 4. If you had two more weeks, what specific next step would you take on the project?
- Given two more weeks, I would cluster users and items based on behavioral features to improve local coherence before applying matrix completion. This allows low-rank assumptions to hold more effectively within each subgroup. Then, I would explore Transformer-based models to capture richer, non-linear interactions between user and item features within clusters. Compared to dot-product factorization, attention mechanisms can better model context-dependent preferences, potentially improving completion quality in sparse, high-dimensional settings.
- 5. If you could restart the project, what is the single biggest change you would make to your approach or plan?
- I'd adopt PyTorch and GPU earlier, and introduce user/item clustering upfront to reduce heterogeneity in the data. This would allow subsequent models to focus on more coherent behavioral patterns, thereby enhancing the granularity and relevance of individual-level predictions.

Thank You

