A Code

We give an example implementation of RANKFROMSETS with the inner product regression function in Eq. (1) in python with the PyTorch package (version 1.1.0). This shows how easy it is to implement RFS; we used a similar implementation to achieve state-of-the-art results in § 3.

```
import torch
from torch import nn
import data

class InnerProduct(nn.Module):
  def __init__(
        self, n_users, n_items, n_attributes, emb_size):
    super().__init__()
    self.user_embeddings = nn.Embedding(n_users, emb_size)
    self.attribute_emb_sum = nn.EmbeddingBag(
        n_attributes, emb_size, 'mean')
    self.item_embeddings = nn.Embedding(n_items, emb_size)
    self.intercepts = nn.Embedding(n_items, 1)

  def forward(
        self, users, items, item_attributes, offsets):
    user_emb = self.user_embeddings(users)
    attr_emb = self.attribute_emb_mean(
        item_attributes, offsets)
    item_emb = self.item_embeddings(items)
    logits = user_emb * (attr_emb + item_emb)
    logits = logits.sum(-1, keepdim=True)
    return logits + self.intercepts(items)

train_data = data.load_data()
learning_rate, batch_size = 0.1, 1000
model = InnerProduct(train_data.n_users,
                     train_data.n_items,
                     train_data.n_attributes,
                     emb_size=100)
optimizer = torch.optim.SGD(
    model.parameters(), learning_rate)
loss = torch.nn.BCEWithLogitsLoss()
# negative samples are in last half of each batch
labels = torch.cat(torch.ones(batch_size // 2),
                   torch.zeros(batch_size // 2))
for batch in train_data:
  model.zero_grad()
  logits = model(*batch)
  L = loss(logits, labels)
  L.backward()
  optimizer.step()
```

# recommendations	10	30	50	70	100
Recall (percent)	0.02	0.06	0.11	0.14	0.21

Table 3: The permutation-marginalized LSTM model performs poorly on the arXiv dataset of researcher reading behavior in terms of in-matrix recall. We report the held-out recall averaged over 100 users due to the computational constraints of the model. It is much lower than that of other methods, but better than random.

B Permutation-invariant models

Collaborative topic Poisson factorization The recommendation model in Gopalan et al. [8] follows a latent variable scheme,

1. **Document model:**
   (a) Draw topics $\beta_{vk} \sim \text{Gamma}(a, b)$
   (b) Draw document topic intensities $\theta_{dk} \sim \text{Gamma}(c, d)$
   (c) Draw word count $w_{dv} \sim \text{Poisson}(\theta_d^T \beta_v)$.
2. **Recommendation model:**
   (a) Draw user preferences $\eta_{uk} \sim \text{Gamma}(e, f)$
   (b) Draw document topic offsets $\epsilon_{dk} \sim \text{Gamma}(g, h)$
   (c) Draw $r_{ud} \sim \text{Poisson}(\eta_u^T(\theta_d + \epsilon_d))$.

Collaborative topic Poisson factorization is a permutation-invariant recommendation model To show that collaborative topic Poisson factorization [8] is permutation-invariant, consider the Poisson likelihood function over words w_{dv}. Conditional on the latent item representation θ_d and latent word representation β_v, every word in the document w_{dv} is independent; the joint probability of words in a document factorizes:

$$p(w_d \mid \theta_d, \beta_v) = \prod_{w_{dv} \in w_d} p(w_{dv} \mid \theta_d, \beta_v). \tag{5}$$

In this model, predictions are made using expectations under the posterior. The posterior is proportional to the the log joint of the model, and the attributes of items (words in documents) enter into the model only via the above product. The product of the probability of words in a document is invariant to a reordering of the words in the document. Therefore, collaborative topic Poisson factorization is permutation-invariant.

C Generalization simulation study

With finite data and a finite number of paramaters, the optimal parameterization of RFS is dependent on the data-generating distribution. Recall that observations of user-item interactions are generated by a Bernoulli distribution with logit function f. We describe a choice of logit function f that leads to the residual and deep architectures in Eqs. (2) and (3) outperforming the inner product architecture in Eq. (1) in terms of predictive performance. Note that the number of parameters across architectures must be equal for a fair comparison. The code required to replicate this experiment is included here:

We simulate data from the following generative process:

1. **For every user** u**:**
   (a) Draw user embedding $\theta_u \sim \text{Normal}(0, \mathbf{I})$.
2. **For every item attribute** j**:**
   (a) Draw attribute embedding $\beta_j \sim \text{Normal}(0, \mathbf{I})$.
3. **For every item** m**:**
   (a) Draw item topics $\theta_m \sim \text{Dirichlet}(\alpha)$
   (b) Draw number of item attributes $M \sim \text{Poisson}(\lambda)$
   (c) Draw nonzero item attributes $x_m \sim \text{Multinomial}(M, \theta_m)$.
4. **For every user, item observation:**
   (a) $y_{um} \sim \text{Bernoulli}\left(y_{um}; \sigma(f(\theta_u, x_m)\right)$

The logit function f is the square kernel:

$$f(\theta_u, x_m) = \left(\theta_u^\top \frac{1}{|x_m|} \sum_{j \in x_m} \beta_j \right)^2 .$$

Before sampling from the Bernoulli, we standardize the logits output by f across users and subtract 7 to achieve sparse user-item observations.

Simulation setup We set the Dirichlet parameter to be $\alpha = 0.01$ and the Poisson rate to be $\lambda = 20$. We generate data for 1k users, 5k item attributes, 30k items, and hold out 100 users for each of the validation and test sets. We fix the momentum to 0.9 [28] and grid search over stochastic gradient descent learning rates of $10, 1, 0.1, 0.01$ and over two learning rate decay schedules. The first linear learning rate decay goes to zero over 100k iterations, while the second divides the learning rate by 10 if the validation in-matrix recall does not improve (evaluation is performed every 500 iterations). We run the grid search on one instance of data generated from this model, then for the best performing hyperparameters for each model trained on this instance, we regenerate data 30 times and average results over these synthetic datasets.

Regression function	Inner product	Deep	Residual
Recall	0.29 ± 0.15	0.32 ± 0.14	0.33 ± 0.18

Table 4: A simulation study demonstrating that the choice of parameterization of RANKFROMSETS is data-dependent. We report the in-matrix recall averaged over 100 users and 30 replications of the simulation where data is regenerated. The residual model in Eq. (3) outperforms the deep model, Eq. (2), and the inner product model in Eq. (1).

Simulation results The results in Table 4 demonstrate that the residual model outperforms both the deep and inner product architectures for data generated by the above generative process.

Generalization The above example shows that the choice of architecture in RFS is data-dependent. To ensure that the model does not overfit as new users or items are included in the training data, we need to compare the number of parameters to the number of datapoints. A model with parameters the size of the training data can overfit by memorizing the training data. For generalization to be possible, overfitting can be avoided if the number of parameters grows slower than the size of the data. The technical backing for this comes from asymptotic statistics and the concept of sieved likelihoods. Specifically, the maximum likelihood estimation procedure with the objective function in Eq. (4) can be replaced by maximization of a sieved likelihood function. The 'sieve' refers to filtering information as the number of parameters (in this case, user and item representations) grows with the number of observations. The sieved likelihood function enables the analysis of asymptotic behavior as the number of users grows $U \to \infty$ and the number of items grows $I \to \infty$, An example of a technique to grow the number of parameters in a way that supports generalization is given in Chapter 25 of Vaart [29].

D Empirical study details

For reproducibility, we describe the hyperparameters used for every model, in addition to qualitative results.

D.1 Recommending research papers

Hyperparameters for RFS We test the stochastic gradient descent algorithm with and without momentum [28]. We use a linear learning rate decay that decays to zero in the maximum number of iterations, 200k. We perform a grid search over learning rates of $1, 5, 10, 15, 25$ and momenta of $0.5, 0.9, 0.95, 0.99$. The minibatch size is set to $\{2^{16}\}$. We use a single negative sample per datapoint, sampled uniformly over the entire dataset. To match the hyperparameters in Gopalan et al. [8], we set the dimensionality of embeddings to 100. Evaluation is performed every 20k iterations.

Hyperparameters for the permutation-marginalized recurrent neural network The embedding and hidden state sizes are fixed to 100. Evaluation is performed every 20k iterations. We grid search over learning rates of $10^{-1}, 10^{-2}, 10^{-3}$ with the Adam optimizer [13]. If validation performance does not improve, we reload the best parameters and optimizer states, and decay the learning rate by 0.9. We subsample single permutations during training and testing.

D.2 Recommending meals

Hyperparameters for RFS. The embedding size is set to 128. For the neural network and residual models in Eqs. (2) and (3) the number of hidden layers is two, and the number of hidden units is set to 256. The item embeddings $g(x_m)$, and item intercepts $h(x_m)$, are computed as the mean of learned food embeddings or intercepts, respectively. We use the RMSProp optimizer in Graves [9] and grid search over learning rates in $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. We use a batch size of 64 and a single negative sample for every datapoint in a minibatch (batch sampling is defined in § 2). Evaluation is performed every 50k iterations.

Hyperparameters for the permutation-marginalized recurrent neural network. We grid search over batch sizes of $\{32, 64, 128\}$. For every item in a minibatch, we sample a single permutation of attributes to approximate the sum over permutations. We use a single negative sample per datapoint, and set the embedding and hidden state sizes to 128. We use the Adam optimizer [13] and grid search over learning rates of $\{10^{-2}, 10^{-3}, 10^{-4}\}$. Evaluation is performed every 1k iterations. If the validation performance does not improve, we reload the best parameters and optimizer states, and decay the learning rate by 0.9.