Click-Through Rate Prediction

Click-Through Rate Prediction

Let's consider an automatic recommendation problem

- Given a set of restaurant indexed on a a web platform (think Tripadvisor)
- ...We want to estimate how likely a user is to actually open the restaurant card

This is know as click-through rate

This example (and the approach) is based on this [TensorFlow Lattice Tutorail]

Loading the Data

Let's start by loading the dataset

In [2]:
```
tr, val, ts = util.load_restaurant_data()
tr.iloc[:3]
```

Out[2]:

	avg_rating	num_reviews	dollar_rating	clicked
0	3.927976	122.0	DDDD	1
1	3.927976	122.0	DDDD	0
2	3.927976	122.0	DDDD	0

- There are two numeric attributes, a categorical one, and a target
- Each row represents one visualization event, hence there might be duplicates

In [3]:
```
dt_in = ['avg_rating', 'num_reviews', 'dollar_rating']
ndup = np.sum(tr.duplicated(dt_in))
print(f'#examples: {len(tr)}, #duplicated inputs {ndup}')
```

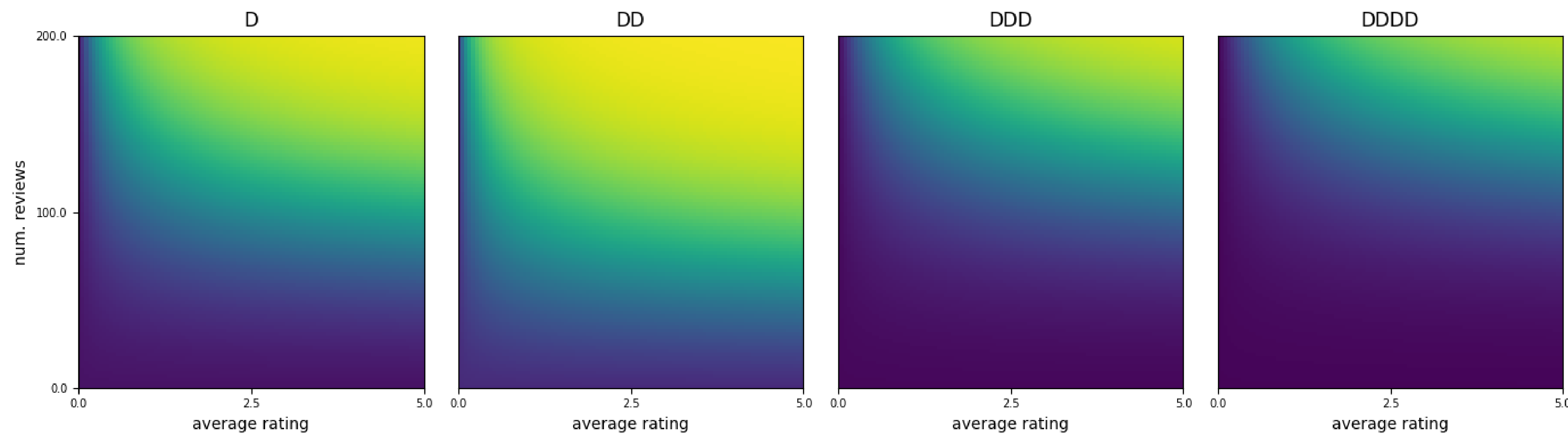
```
#examples: 835, #duplicated inputs 395
```

- The click rate can be inferred by number of clicks for each restaurant

Target Function

This is a synthetic dataset, for which we know the target function

```
In [4]: util.plot_ctr_truth(figsize=figsize)
```

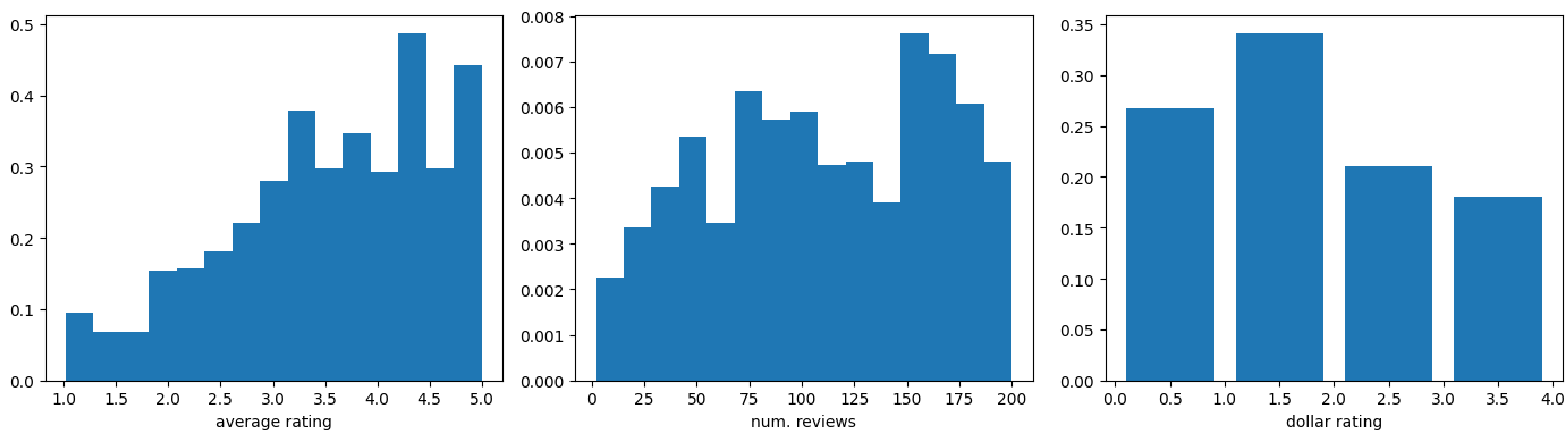


- The click rate grows with the average rating and the number of reviews
- Average priced restaurant are clicked the most

Data Distribution

Let's check the attribute distribution on the training set

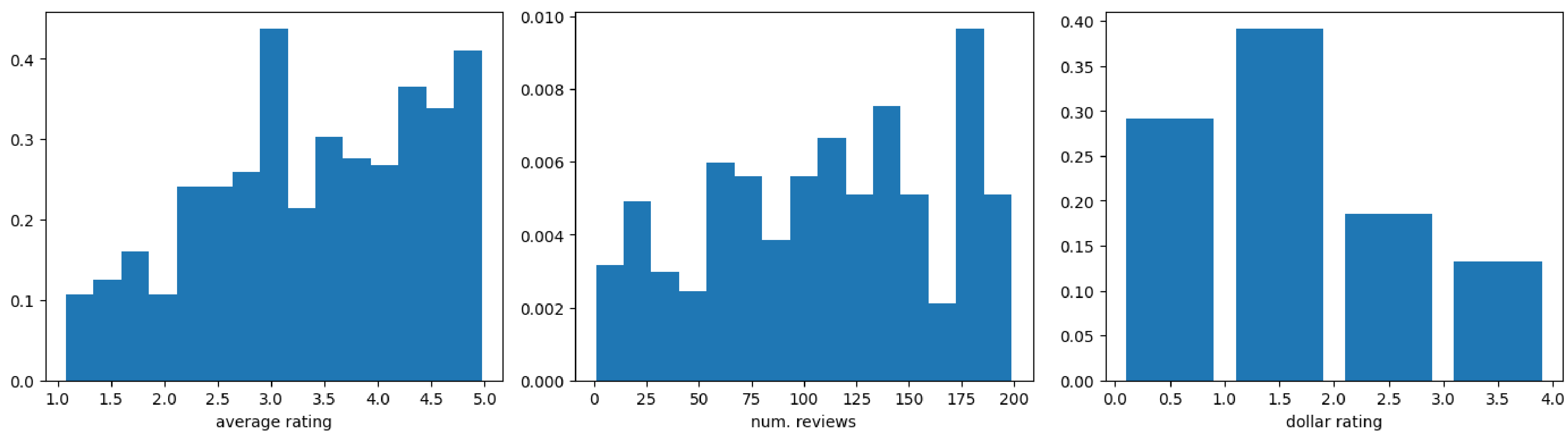
```
In [5]: util.plot_ctr_distribution(tr, figsize=figsize)
```



Data Distribution

...Then on the validation set

In [6]:
```
util.plot_ctr_distribution(val, figsize=figsize)
```

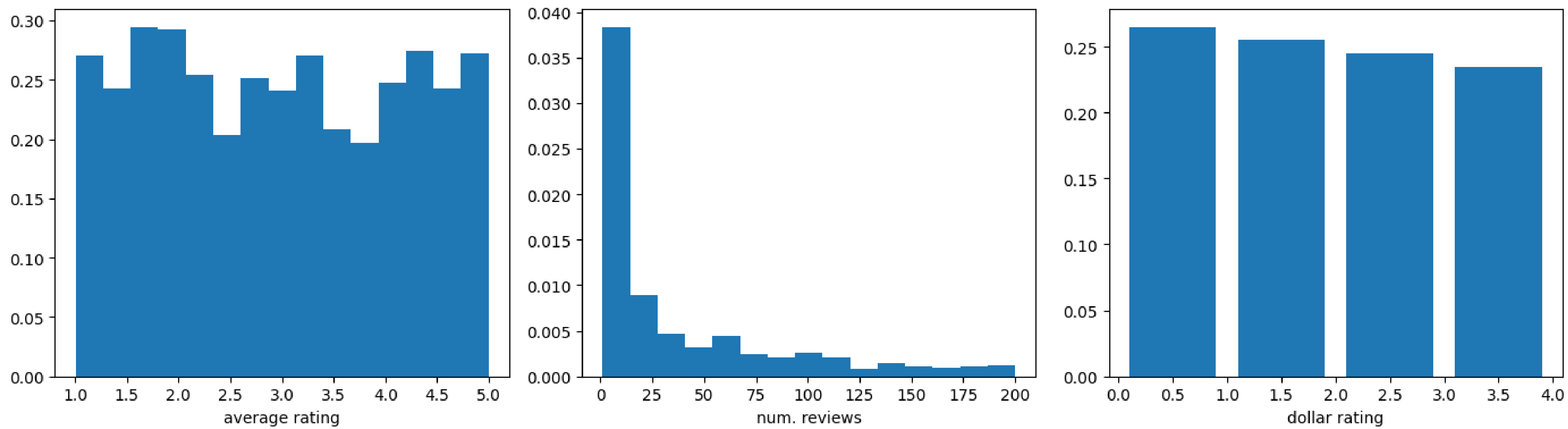


Not exactly the same, but it roughly matches

Data Distribution

...And finally on the test set

```
In [7]: util.plot_ctr_distribution(ts, figsize=figsize)
```



Here there is a strong discrepancy w.r.t. the training set

Distribution Discrepancy

What is the reason for the discrepancy?

A training set for this kind of problem will come from app usage data

- Users seldom scroll through all search results
- ...So their clicks will be biased toward high ranked restaurant

Any training set obtained in this fashion will be strongly biased

However, click rate prediction is typically use for ranking search results

...Meaning that we will need to evaluate also less viewed restaurants

- In a practical problem, the test set would not even be available
- We have it just as a mean for validating our results

A bias in the training can be problematic: we will try to see that in action

How would you deal with this problem?

How would you deal with this problem?

Using sample weights (or data augmentation) might be a solution
...But here we will focus on a different angle

A Baseline Approach

Preparing the Data

We will start by tackling the problem using a Multi Layer Perceptron

We normalize the numeric data:

In [8]:
```
nf = ['avg_rating', 'num_reviews']
scale = tr[nf].max()

tr_s = tr.copy()
tr_s[nf] = tr_s[nf] / scale
val_s = val.copy()
val_s[nf] = val_s[nf] / scale
ts_s = ts.copy()
ts_s[nf] = ts_s[nf] / scale
```

We also adopt a one-hot encoding for the categorical data:

In [9]:
```
tr_sc = pd.get_dummies(tr_s).astype(np.float32)
val_sc = pd.get_dummies(val_s).astype(np.float32)
ts_sc = pd.get_dummies(ts_s).astype(np.float32)
dt_in_c = [c for c in tr_sc.columns if c != 'clicked']
```

Preparing the Data

Here is the result of our preparation

In [10]:
```
tr_sc
```

Out[10]:

	avg_rating	num_reviews	clicked	dollar_rating_D	dollar_rating_DD	dollar_rating_DDD	dollar_rating_DDDD
0	0.785773	0.610	1.0	0.0	0.0	0.0	1.0
1	0.785773	0.610	0.0	0.0	0.0	0.0	1.0
2	0.785773	0.610	0.0	0.0	0.0	0.0	1.0
3	0.866150	0.610	1.0	0.0	0.0	0.0	1.0
4	0.619945	0.590	0.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...
830	0.597304	0.055	1.0	0.0	1.0	0.0	0.0
831	0.783784	0.505	1.0	1.0	0.0	0.0	0.0
832	0.783784	0.505	1.0	1.0	0.0	0.0	0.0
833	0.688336	0.270	1.0	0.0	1.0	0.0	0.0
834	0.688336	0.270	0.0	0.0	1.0	0.0	0.0

835 rows × 7 columns

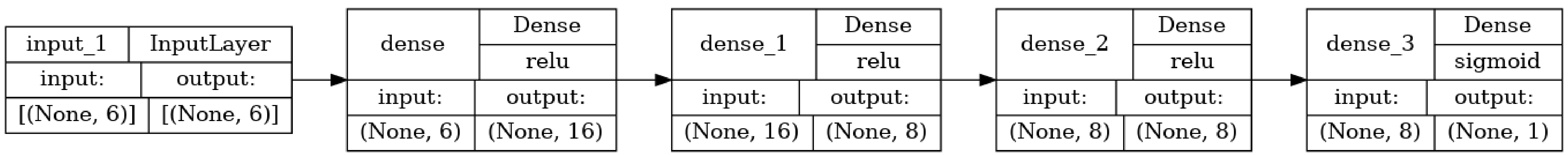
Building a Baseline Model

Let's start by ignoring the issue

...And building as baseline model an MLP classifier

In [11]:
```
nn = util.build_nn_model(input_shape=len(dt_in_c), output_shape=1, hidden=[16, 8, 8], output
util.plot_nn_model(nn)
```

Out[11]:

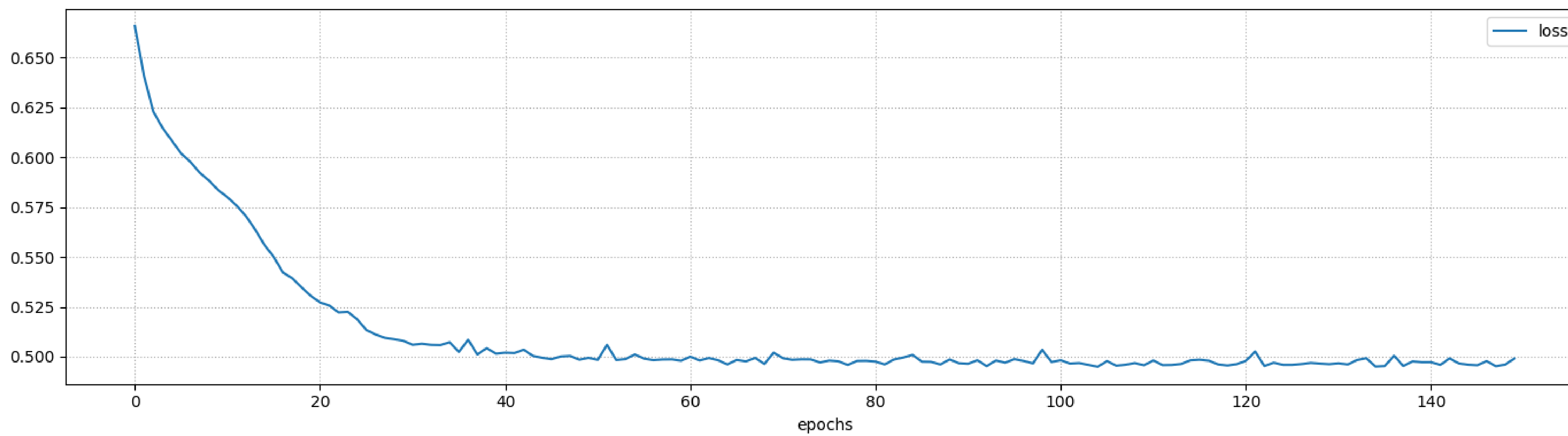


- Remember this is a stochastic prediction problem
- So, even if we train a classifier we are not interested in classes
- Rather, we care about estimated probabilities

Training the Baseline Model

We can train the model as usual

```
In [12]: nn = util.build_nn_model(input_shape=len(dt_in_c), output_shape=1, hidden=[16, 8, 8], output
         history = util.train_nn_model(nn, tr_sc[dt_in_c], tr_sc['clicked'], loss='binary_crossentrop
         util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.4991 (training)
```

- It seems we are reasonably close to convergence

Evaluating the Predictions

This is not a classification problem, so accuracy is not a good metric

- The output of our system is meant to be interpreted as a probability
- ...So, rounding to obtain a deterministic prediction may be too restrictive

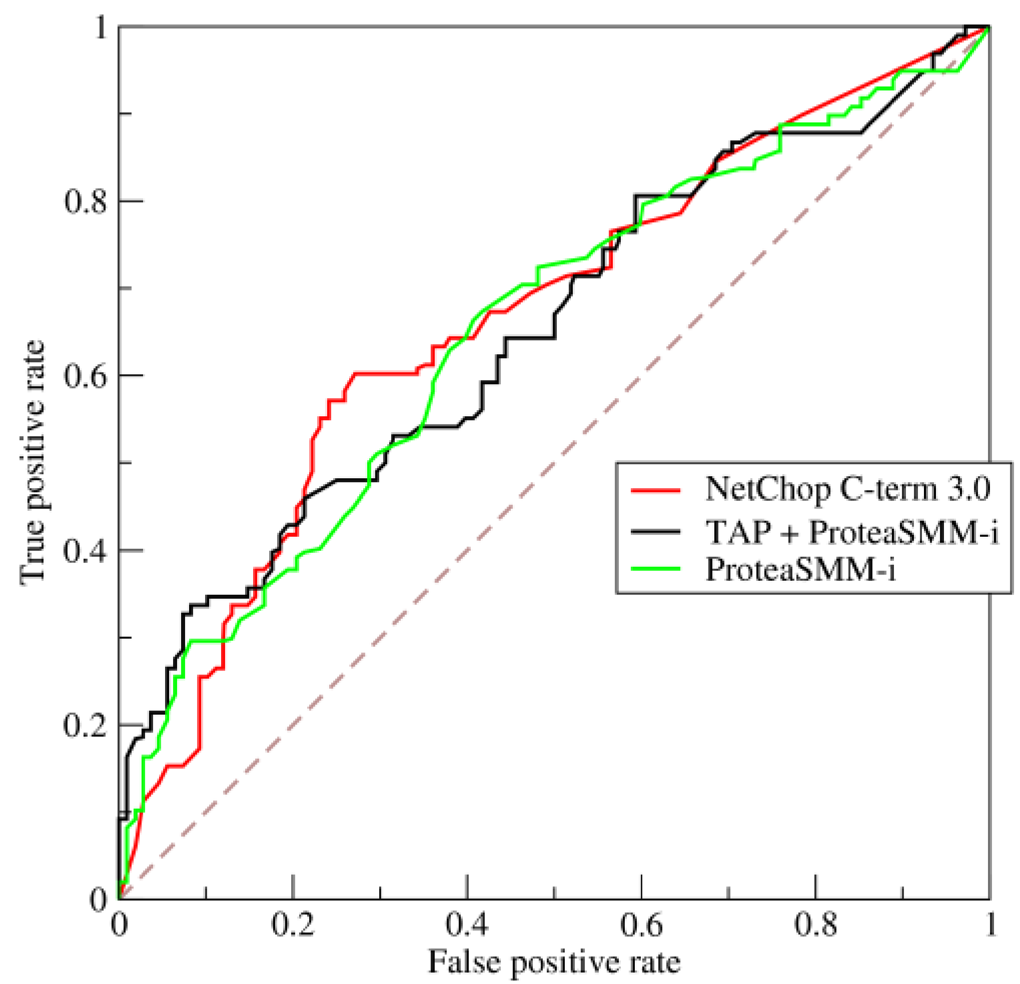
Instead, we will make a first evaluation using a ROC curve

A Receiver Operating Characteristic curve is a type of plot

- We consider multiple threshold values
  - Each threshold is meant to be used for discriminating between classes
  - The usual rounding approach is equivalent to a 0.5 threshold
- On the x axis, we report the false positive rate for each threshold
- On the y axis, we report the true positive rate for each threshold

Evaluating the Predictions

A ROC curve looks like this (image from wikipedia)



- The large the Area Under Curve (AUC), the better the performance
- The AUC value is guaranteed to be in the $[0, 1]$ interval

Evaluating the Predictions

Let's compute the AUC values for all sets

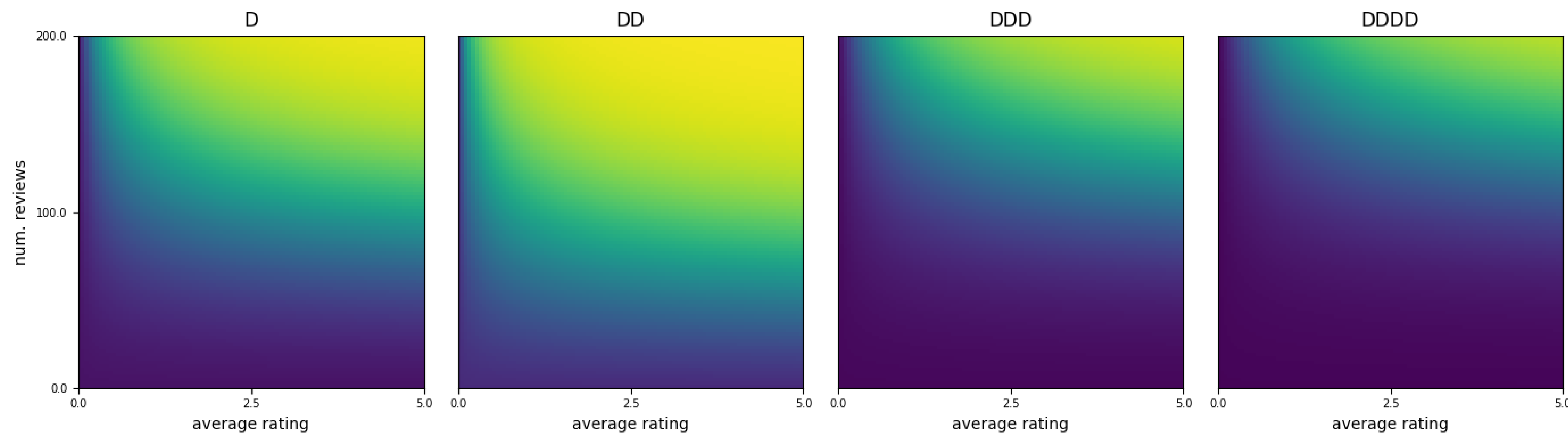
```
In [13]: pred_tr = nn.predict(tr_sc[dt_in_c], verbose=0)
         pred_val = nn.predict(val_sc[dt_in_c], verbose=0)
         pred_ts = nn.predict(ts_sc[dt_in_c], verbose=0)
         auc_tr = roc_auc_score(tr_sc['clicked'], pred_tr)
         auc_val = roc_auc_score(val_sc['clicked'], pred_val)
         auc_ts = roc_auc_score(ts_sc['clicked'], pred_ts)
         print(f'AUC score: {auc_tr:.2f} (training), {auc_val:.2f} (validation), {auc_ts:.2f} (test)'

         AUC score: 0.81 (training), 0.80 (validation), 0.76 (test)
```

- The model works well on the training distribution
- But less well on the testing data (as expected)

Issues with the MLP

Here we have again the ground truth for our click rate

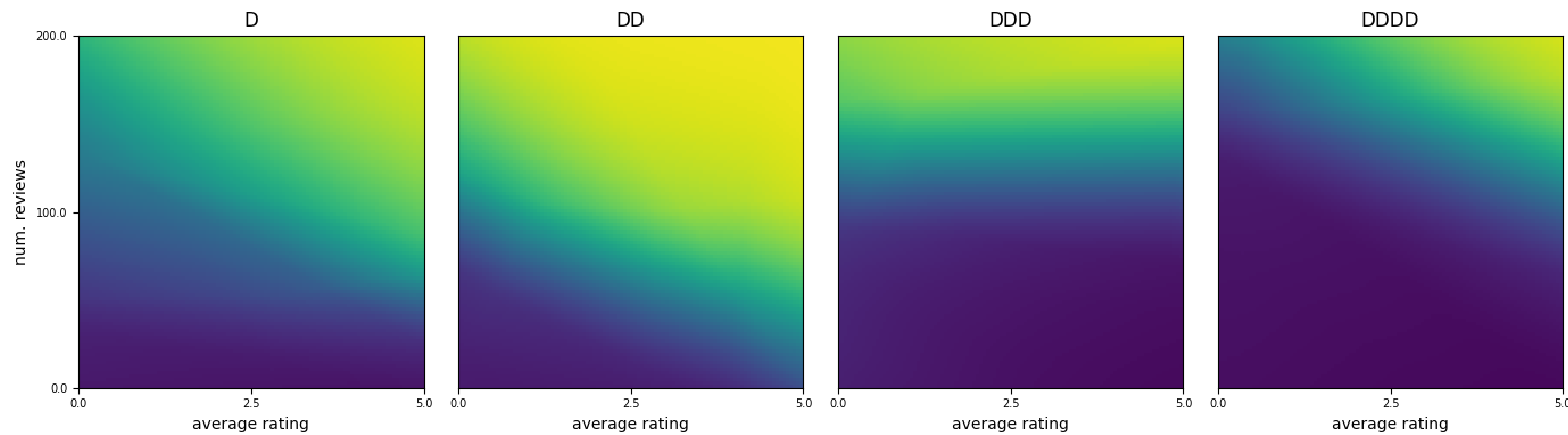
```
In [14]: util.plot_ctr_truth(figsize=figsize)
```



Issues with the MLP

...And here is the full (prediction) output space for the MLP

```
In [15]: util.plot_ctr_estimation(nn, scale, figsize=figsize)
```



Something odd is happening here: can you tell what?

Constraint Violations

In some areas, increasing an attribute has the opposite of the expected effect

Our problem has natural monotonicities, which are may not hold for the MLP

- The motivation is that poor data for some region of the input space
- ...And ML models often have poor out-of-distribution behavior

This is a significant issue in practice

Having a statistically representative training set is a luxury

- E.g. time series, organ trasplants programs, promo sales...
- If we give up on those problem, we loose a lot of potential

Sample weights cannot fix this issue

- In fact, most ML models are naturally incapable of enforcing constraints
- ...But most is not all, so let's see some exceptions :-)