CS 536 : Final Project - Data Completion and Interpolation

Haoyang Zhang, Han Wu, Shengjie Li

May 11, 2019

1 Introduction, group members and division of workload

In this group project, we implemented an autocoder for interpolating missing features from features we have and achieved .

Name NetID	Workload
Han Wu hw436	Fine-tuned the parameters of our model. Did some experiments for the evaluation of our model. Wrote part of the report.
Haoyang Zhang hz333	Analyzed and wrote scripts to clean the data. Wrote scripts to restore human-friendly data from the output of our model. Wrote part of the report.
Shengjie Li sl1560	Implemented the basics of neural networks including back-propagation and several loss functions and activation functions. Wrote part of the report.

2 Data preprocessing and thoughts towards this project

1. How to represent or process the data. Data features may contain a number of diverse data types (real values, integer values, categorical or binary values, ordered categorical values, open/natural language responses). How can you represent these for processing and prediction?

   - **Data encoding**

     Given the `ML3AllSites` dataset, we can classify each column into 6 types: (positive) integers, unordered multiple choices, Boolean-like values, long texts, other valid responses and NA's.

     (Note: the 1177th row is wrongly encoded in the original dataset, but we fixed it in the `ML3AllSitesC`. However, it again includes some more bugs in date related columns like column 124 Date.x because of Microsoft Excel. Anyway, we eventually fixed them when reading the dataset when calling `dataFormat.py`)

     A detailed codebook is available in Appendix A, and further reference is in `/code/dataFormat.py`, which is organized in the order of original dataset.

     - (Postive) Integers

       Many columns belong to this type, for examples, best grade 1 (column 21), mcdv1 (column 71), temperature in lab (column 126) and intrinsic (column 247). If the choices are exactly ordered, say "1" is "unhappy", and "10" is "happy", we consider this column as this type. Tough "2" may not be twice happier than "1", nevertheless, 2 is indeed happier. Note that some columns may include float numbers, say intrinsic, but we can multiply a factor to scale all responses to integers. Also, some

responses could be negative values, say mcdv1, but we still can add a number to shift all of them to non-negative values.
  Each value in these columns is the real value instead of probability.

  - Unordered Multiple Choices
    Many columns belong to this type, for examples, ethnicity (column 42), gender (column 44), major (column 70) and V position (column 115). Note that these columns may include natural language response, say major, but we have classified all responses into several choices.
    Each choice of these columns are exclusive and unordered. Therefore, we cannot just simply encode them into integers. Or we will have to face the explanatory problem: If we encode "computer science" into "1" and "mathematics" into "2", do we mean a "mathematics" is equal to 2 "computer science"? Therefore, we choose one-hot encoding to use the same number as choices of Boolean values to represent the participant's choice. In this case, we can consider each value as the probability that this participant will choose this response.
  - Boolean-like Values
    A few columns belong to this type. Some are natural language responses but there is a true answer, say anagrams (column 5 and 6) and attention correct (column 10), and the test is highly concerning about whether the participant correct or not instead of what they answered. Others are multiple choices with exactly 2 possible answers like mcmost (column 76 to 80), and for simplicity we prefer to use 1 Boolean value to represent his/her choice.
    In general, we can consider this type as a special multiple choice type. Namely, each value in this column is a probability.
  - Long Texts
    There are exactly 3 columns belongs to this type: highpower (column 45), lowpower (column 67) and Notes (column 134). Because of time limition, we skiped to process these 3 columns.
  - Other Valid Responses
    Some natural language responses that describe a real number belong to this type, for example, K ratio (column 66), worst grade 2 (column 118) and SR TF Correct (column 133). Some obviously unrelated or redundant data is also this type. For an example, Date Computer (column 220) is a duplicate to Month Computer, Day Computer and Year Computer (column 222 to 224).
  - NA's
    In order to distinguish normal data and NA's, we use a valid mask. For each encoded feature, we use a Boolean value to indicate whether it is normal or NA. Namely, each row in original dataset is coded into 2 rows, where one is a valid mask and the other is the real data.
    For simplicity, we set all NA's to 0 just like dropout. When computing error, we use the mask to set these features' loss to 0.

- **Data preprocessing**
  - Scaling to $[0, 1]$
    Notice that the coded data can be further devided into 2 types: real values and probabilities. Notice that real values could be from negative infinity to positive infinity. (There may be some more restrictions like temperature in lab cannot be lower than -460. But in general its range is way larger than $[0, 1]$.) But for our prediction simplicity, we will linearly scale the largest value seen to 1 and the smallest to 0.
    Here are some more things we can do, but because of time limit, we skipped them. The straightforward problem for this naïve scaling is we might be trapped by

outliers. For example, (this example is already fixed.) some participant claimed his/her/its age is about 150. If we directly apply the scaling, most 20-ish responses will be scaled into about 0.007, and the only response that is greater than 0.3 is that 150, which will make this feature hard to predict precisely. Therefore, we should throw out these outliers.
But a further thought is that this situation can also happen when the response distributed unevenly. For example, many people answer either about 1 to 2 or 8 to 9. In this case, we use a lot of space to encode unlikely values, which leads to the same result. In this case, a nonlinear scale method will be helpful. A rudimentary thought is we sort all values in the dataset and linearly scale first 10% values to the range $[0, 0.1)$, second 10% to $[0.1, 0.2)$, and so on.
Another problem is that we cannot predict any larger or smaller values than values in the dataset. A plausible justification could be that it is generally unlikely to see an extreme small or large value. But if we adopt the nonlinear scale method, we can map negative infinity (or the smallest valid value) to the smallest value in dataset into $[0, 0.1)$, and all values in dataset to $[0.1, 0.9)$, and so on.

- Grouping Features
  Notice that this dataset is all about 10 psychological tests. Therefore, we can assume the features in the same tests are more related than features between different tests. Namely the dimensionality in each tests is relatively small. Therefore, we can group features in terms of tests. Some global information about this participant, like demographic features and personality features, is grouped into another global set instead of test sets, which is called group 0. Some definitely unrelated data like participant ID (column 1) is grouped into another set, called group 11.
  In this way, we can try to use group 0 and each test group to predict blank features in this test just by picking 2 group indices instead of a huge number of feature indices.
- Selecting Features
  Notice that all group 11 features can be discard based on our prior knowledge. (Actually we should use graphical model to prove it.) A further thing we should do is run Chow Liu Algorithm on group 0 and each test group to find weak-dependent intra-test features (Namely, all its edges are weak.) and eliminate them, and then run it on the whole feature space to try to further eliminate features.
- Discarding Almost NA Data Points
  Notice that there are several almost blank rows in the original datasets. These data points cannot tell us many things. We can use NA masks to identify them. More specifically, we remove these rows whose mask has more than 100 `False`.

2. How to model the problem of interpolation. What are the inputs, what are the outputs? An important if subtle question to consider here - what does it mean to predictor or interpolate a missing feature?

   Given the preprocessed data, each data point is a vector with some blanks, and the whole dataset is a matrix with blanks. Our goal is to fill the blanks. Notice that we can do this because the dimensionality of this matrix is limited. Namely, many features are related to each other. For example, those who claimed they are high self-esteemed (column 252) are generally less stressed (column 253) and their mood (column 248) is better. Therefore, we can consider the dataset as a limited-rank matrix with some blanks whose size is 2434×261.

   A straight forward idea to this problem is consider the blanks as noise, and our goal becomes to detect and eliminate this noise. Notice that the rank of this matrix is limited, therefore we can transform it into a much smaller matrix and restore it. Assuming the noise is relatively smaller than the information that this matrix gives, when we are transform or compressing this matrix, the noise will be eliminated, and then we can decompress it to restore the blanked values.

3. Model selection. What kind of model or models do you want to consider?

   Before we started doing this project, we discussed several models.

   - Autoencoders

     It can compress and decompress the given inputs into smaller representations, and these representations contain key information of the inputs. Thus, we can make use of these representations to classify data and generate data, which makes autoencoders handy in this situation.
   - A transfer learning approach:

     First, we can train an autoencoder to compress information from the inputs, minimize the reconstruction loss. After we finish the training of the autoencoder, we discard the decoder, and use the encoder to get representations of data points. Then, we train a classifier on top of the representations.
   - Autoencoders with RNN:

     Since the data contains natural language data, we could use RNN to deal with this kind of data.

3 Requirements

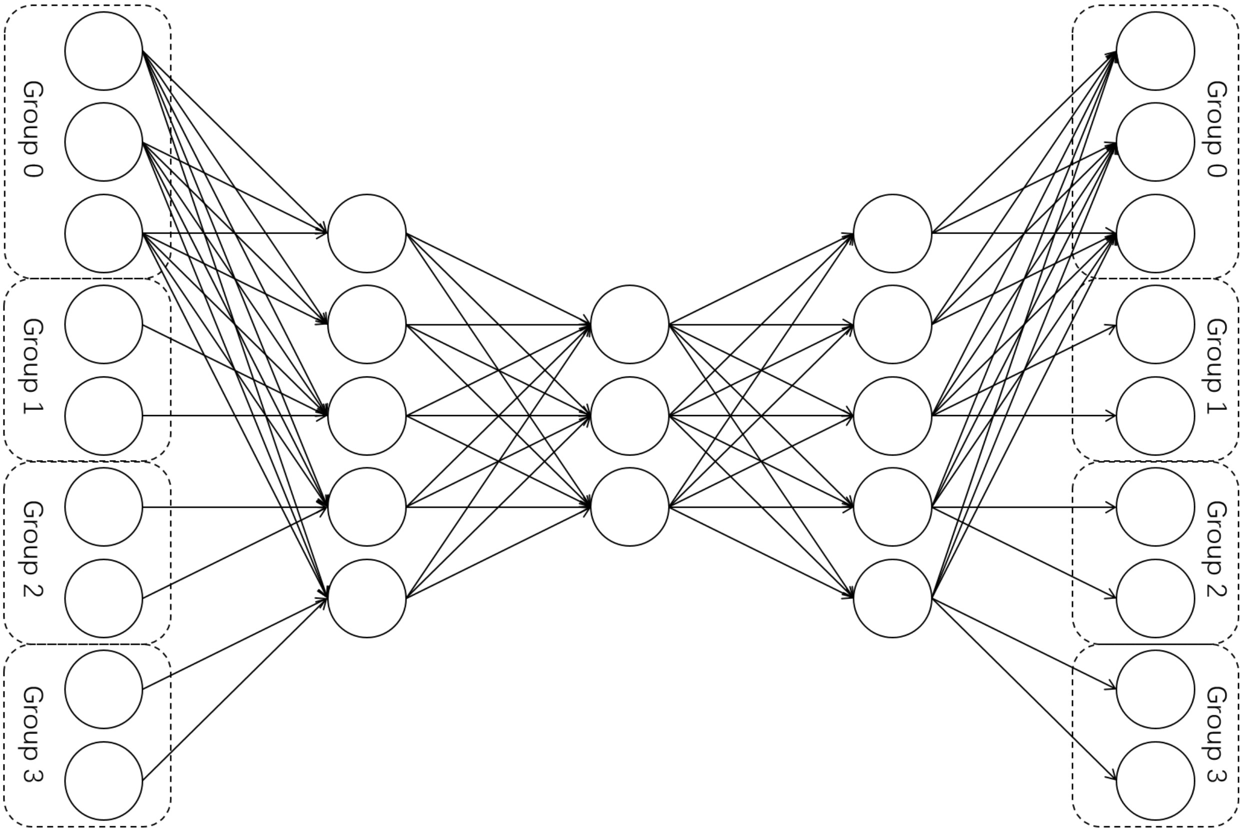


Figure 1: Our autoencoder

1. **Describe your Model:** What approach did you take? What design choices did you make, and why? How did you represent the data? How can you evaluate your model for goodness

of fit? Did you make an effort to identify and exclude irrelevant variables? How did you handle missing data?

Given the Formalization of interpolation, we eventually chose to use an autoencoder as our model because it can compress and decompress the given inputs and it is not complicated to build.

- Structure
  Figure 1 shows the structure of our model.
  (a) The input
  There are 223 nodes in the input. We scaled the input to be in $[0, 1]$. Each node could be either in a classification group or in a real value group. The activation function here is ReLU in order to make it less computationally expensive. The dropout rate of this layer is 10%.
  (b) Hidden layer 1
  There are 180 nodes in this layer. The activation function here is ReLU.
  (c) Hidden layer 2
  There are 100 nodes in this layer. The activation function here is ReLU.
  (d) Hidden layer 3
  There are 180 nodes in this layer. The activation function here is ReLU.
  (e) The output
  There are 223 node in the output. The activation function here is Sigmoid in order to make values lie in $[0, 1]$.
- Loss/Error
  We are measuring the reconstruction errors. For the reason that there are many data types, we are treating the data over different loss functions.

  For real values and integer values, we are using mean squared error (m denotes the number of data points)

$$L_{MSE}(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\underline{y}^i - \underline{p}^i)^2,$$

  and root-mean-squared error

$$L_{RMSE}(\theta) = \sqrt{L_{MSE}(\theta)}.$$

  For categorical values, we first converted them to one-hot encoding, then we are treating the problem as a multi-label classification problem (for these categorical values only). Thus, for categorical values, we are using binary cross-entropy loss

$$L_{BCE}(\theta) = \frac{1}{m}\sum_{i=1}^{m}[\underline{y}^i \log(\underline{p}^i) + (1 - \underline{y}^i)\log(1 - \underline{p}^i)]).$$

  Because the data set is a combination of different types of data, we are using a combination of different loss functions.
- Data
  This is included in Data preprocessing. We are only training over part of the features. For real values data, we scale it into $[0, 1]$. For categorical data, we used one-hot encoding to represent which also make the data lie in $[0, 1]$. Basically all data are in $[0, 1]$ and we treated all of them as float numbers.

We manually removed some irrelevant features such as participant ID. We don't think we are able to train over these features.

When dealing with missing features, we have a mask to let us know which feature is missed and then we don't need to calculate loss on this feature.

- Other facts

  Since we already grouped features as Figure 1 shows, we can first encode each groups and then encode the whole data points. In this way, we can dramatically eliminate the number of weights because our first layer is not dense.

  Notice that we encode the multiple choice columns into several features. Therefore, we can append a softmax layer to each multiple choice column. One thing to notice that for non-choice features, we have to directly output its value instead of pass it to softmax layer, or it will always return 1. (When we are writing this report, we notice that we can expend each non-choice features into 2 features x and $\bar{x} = 1 - x$. In this case, we can directly pass all features of the same column to a softmax, instead of using masks to identify non-choice features.)

  Here is another interesting but not necessarily useful fact. Notice that we assumed the rank of this matrix is limited. Namely the transformation can actually be linear. Namely we literally can multiply 2 small matrices to get this matrix. Therefore, we can further define this problem as find 2 matrices A and B to minimize $||M - AB||$, which is literally an optimization problem. But the drawback is that we ignore the meaning of each feature, say sum of several features should always be 1. But it might be not a bad start, and we can further use Ada boost to combine it with our AutoEncoder.

2. **Describe your Training Algorithm:**

   We implemented a gradient-based optimizer–stochastic gradient descent with back-propagation for the training part. Comparing with other types of gradient descent, stochastic gradient descent uses one data point at a time, which is computationally friendly. We are only training over 223 features. Our training process was actually quite fast, so we didn't do any compromises.

3. **Describe your Model Validation:**

   We divided the whole data set into 3 data sets: train set, dev set and test set. During the training process, we measure loss on train set and dev set. Dev set is an indicator of fitness and has been used in fine-tuning of the hyperparameters. If dev loss starts to climb up, we could know it is overfitting. After we finish the training of the model, we evaluate loss on test set.

   For the prevention of overfitting, we are using a simple architecture, and we added drop-out rates on layers to enable the model to 'disable' some neurons for some probabilities. Because we are dealing with missing data and trying to interpolate missing data, drop-out makes the model be used to incomplete input data. Since the dataset isn't massive, drop-out 'creates' many incomplete data point, which boosts the size of the dataset.

   After the training, our model achieved a low loss of 20-ish.

4. **Evaluate your Model:**

Feature selection is a very important issue in this project. There are many features in the dataset. Some of them are useful. They play a great role in predicting the missing data. Some of them are useless. They are meaningless data under the context of predicting missing data in the dataset. We first select some potential unimportant feature groups. Then, we set parts of them to zero, and watch if the test loss increases. If the test loss increases much, then these features are important. If the corresponding test loss does not increase much, then this group of features is not important.

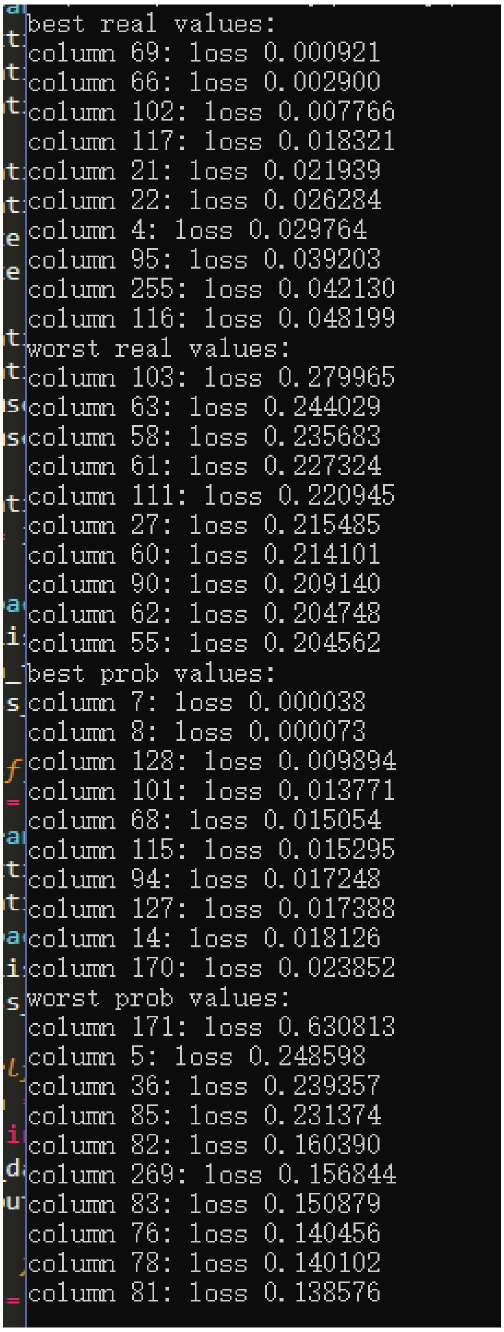
We set the number of neurons in the second and third layers to be 180 and 100. We set dropout to be 0.1 because it is faster and more appropriate in this experiment.

There are 9 potential unimportant groups. They are 120:130, 130:150, 150:164, 164:175, 175:182, 182:189, 189:194, 194:205, and 205:223. We use a vector to denote whether to close or open a group of features. 0 represents closing a group, setting none of the features in this group to 0. 1 represents opening a group, which means setting all of the features in this group to 0. 000000000 represents close all groups, which means to keep the original value and set none of them to 0. 000000000 is the baseline case.

The experimental results are shown below.

Close-open combination	Test error	Train error	epoch
100000000	27.2508	25.2822	30
010000000	29.7388	26.3616	60
001000000	25.8937	22.2732	50
000100000	28.4796	26.0596	25
000010000	26.9821	23.3587	45
000001000	25.6469	23.2698	25
000000100	25.9523	23.8008	25
000000010	25.4880	22.3731	40
000000001	26.5137	22.9699	60

From the experiment, we can see that features 194:205 is the most unimportant features. Their test error is the smallest, which means opening them results in the smallest error increasing. So, they are the most useless features in predicting the missing data. We map this intermediate features into the original feature space. In the original feature space. Features 21- 25, 104, 117-121 are the most unimportant features. We can ignore them to decrease the computational complexity in real-world applications.



```
best real values:
column 69: loss 0.000921
column 66: loss 0.002900
column 102: loss 0.007766
column 117: loss 0.018321
column 21: loss 0.021939
column 22: loss 0.026284
column 4: loss 0.029764
column 95: loss 0.039203
column 255: loss 0.042130
column 116: loss 0.048199
worst real values:
column 103: loss 0.279965
column 63: loss 0.244029
column 58: loss 0.235683
column 61: loss 0.227324
column 111: loss 0.220945
column 27: loss 0.215485
column 60: loss 0.214101
column 90: loss 0.209140
column 62: loss 0.204748
column 55: loss 0.204562
best prob values:
column 7: loss 0.000038
column 8: loss 0.000073
column 128: loss 0.009894
column 101: loss 0.013771
column 68: loss 0.015054
column 115: loss 0.015295
column 94: loss 0.017248
column 127: loss 0.017388
column 14: loss 0.018126
column 170: loss 0.023852
worst prob values:
column 171: loss 0.630813
column 5: loss 0.248598
column 36: loss 0.239357
column 85: loss 0.231374
column 82: loss 0.160390
column 269: loss 0.156844
column 83: loss 0.150879
column 76: loss 0.140456
column 78: loss 0.140102
column 81: loss 0.138576
```

Figure 2: Loss of each feature to the original data

5. **Analyze the Data:**

   Are there some features it is really good at predicting and some it is really poor at predicting? Why do you think that is?

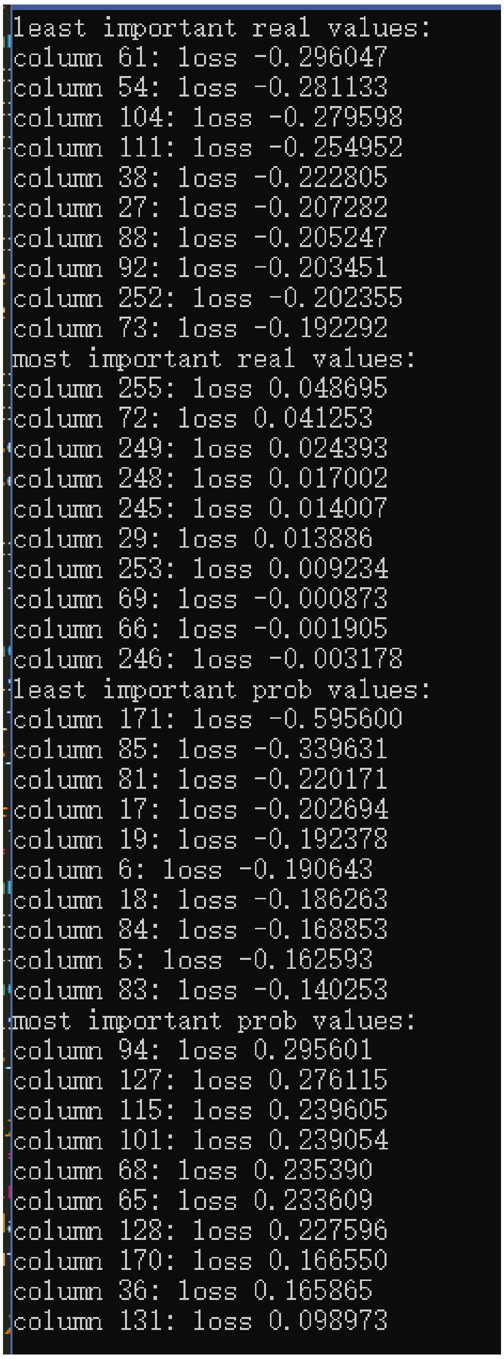
   In order to answer this question, we traced the loss of each feature to the original data, and the result is shown in Figure 2:

   Notice that the best real-value feature we predict is L ratio (column 69), and it is really diverse. A plausible explanation is that this effect is “strong” enough, and it is true according to the original study (ML3). Actually, the best 3 features (L, K, R ratio) all belong to this effect.

   The worst real-value feature is sarcasm (column 103). Notice that this effect includes a number of natural language response which we preprocessed roughly, and this effect is also not that significant. Therefore, it is reasonable we cannot predict this feature well.

Column 7 and 8 are anagrams 3 and 4, which we encode any word into "correct answer". Therefore, it should be the best 2 features because it can literally always predict "correct answer". Except these 2 feature, the best probability feature is clipboard weight (column 128), it is interesting because there is no directly related feature except the participant answers. (Note that the clipboard material is independent to its weight.) Namely, we do find that the weight of clipboard will influence participant's response.

The worst probability feature is persistence. (It is a bug actually because I preprocessed it into a probability feature but it actually is a real value. It is my fault. But a good story is it shows we should not use cross entropy to measure the loss of a real-value feature.) The worst feature except for that bugged one is anagrams 1 (column 5). A plausible explanation is that this feature is sort of coincident because we may either happen to have the "Eureka" time instantly or be puzzled for a long time and quit this question.



```
least important real values:
column 61: loss -0.296047
column 54: loss -0.281133
column 104: loss -0.279598
column 111: loss -0.254952
column 38: loss -0.222805
column 27: loss -0.207282
column 88: loss -0.205247
column 92: loss -0.203451
column 252: loss -0.202355
column 73: loss -0.192292
most important real values:
column 255: loss 0.048695
column 72: loss 0.041253
column 249: loss 0.024393
column 248: loss 0.017002
column 245: loss 0.014007
column 29: loss 0.013886
column 253: loss 0.009234
column 69: loss -0.000873
column 66: loss -0.001905
column 246: loss -0.003178
least important prob values:
column 171: loss -0.595600
column 85: loss -0.339631
column 81: loss -0.220171
column 17: loss -0.202694
column 19: loss -0.192378
column 6: loss -0.190643
column 18: loss -0.186263
column 84: loss -0.168853
column 5: loss -0.162593
column 83: loss -0.140253
most important prob values:
column 94: loss 0.295601
column 127: loss 0.276115
column 115: loss 0.239605
column 101: loss 0.239054
column 68: loss 0.235390
column 65: loss 0.233609
column 128: loss 0.227596
column 170: loss 0.166550
column 36: loss 0.165865
column 131: loss 0.098973
```

Figure 3: feature importance

What features were particularly valuable in predicting/interpolating? What features weren't particularly useful at all?

In order to answer this question, we dropout each feature and calculating the loss after we dropout it. In case of the situation that this feature itself is hard to predict, when we calculating the loss, we will also ignore this feature. The result is in Figure 3.)

Notice that most of the most important real-value features are aggregated features like NFC, Mood and Agreeableness. It is reasonable because they tell us a whole aspect of this participant, and the response of this participant can be consider as a function of all aspects. (Refer to our transfer learning thought.) Similarly, we can find that most of the least important real-value features are the actual response of each aggregated feature, like intrinsic_01, self-esteem_01 and soon. Since they have been aggregated into another features, they do tell us less.

Notice that order of tasks (column 127) does important. It corresponds to the original study design. Some other important features are N position and V position. It is also reasonable because it will determine whether the next response will be greater than 10 or less than 10.

Again, because of my fault, persistence (column 171) become the least important feature. Except for this, the back count features are generally less important. A justification is most participant will correctly answer this question.

6. **Generate Data:**

   There are two approaches:

   (a) We picked a complete datapoint and removed part of it, used this to predict the removed features. Then we compared the predicted features and the original features. If they are similar, then we say it's a successful generation.

   Our model was quite good at this job.

   (b) To generate a new data point, first we picked a complete data point, then we removed part of it and use the removed data point to predict the removed part. Then we combined the removed data point and the prediction and repeat this process. At last all data from the original data point was removed.

   To determine whether it's good, we compare the new data point with the original data point. If they are similar, then we say it's a successful generation.

   Our model was doing alright at this job.

4 Bonus

Appendix A Codebook

Codebook

This part describes how we encode the original datasets. Each row is `column {original column} ({coded start column}: {coded end column}): data{column name}: {original column dtype} / {preCoded dtype} / {coded dtype}`. `original column` is the column in original datasets. `coded column` is the column in coded dataset that inputs our model, and it includes `start column` but excludes `end column`. `column name` is the name in original dataset (according to its first row). `original column dtype` is the data type in original dataset. `preCoded dtype` is the middle state data type. `coded dtype` is the coded data type before scaling.

Some column that we discared is set to `None`. Note that column 21 and 117 is further coded into `int` when we are scaling them.

```
column 0 (223: 244): dataSite: str / int / [bool’s]
column 1 (244: 245): dataParticipant_ID: str / int / int
column 2 (245: 246): dataRowNumber: str / int / int
column 3 (246: 247): datasession_id: str / int / int
column 4 (0: 1): dataage: str / int / int
column 5 (150: 151): dataanagrams1: str / int / bool
column 6 (151: 152): dataanagrams2: str / int / bool
column 7 (152: 153): dataanagrams3: str / int / bool
column 8 (153: 154): dataanagrams4: str / int / bool
column 9 (1: 7): dataattention: str / int / [bool’s]
column 10 (130: 131): dataattentioncorrect: str / int / bool
column 11 (164: 165): databackcount1: str / int / bool
column 12 (165: 166): databackcount10: str / int / bool
column 13 (166: 167): databackcount2: str / int / bool
column 14 (167: 168): databackcount3: str / int / bool
column 15 (168: 169): databackcount4: str / int / bool
column 16 (169: 170): databackcount5: str / int / bool
column 17 (170: 171): databackcount6: str / int / bool
column 18 (171: 172): databackcount7: str / int / bool
column 19 (172: 173): databackcount8: str / int / bool
column 20 (173: 174): databackcount9: str / int / bool
column 21 (194: 195): databestgrade1: str / (int, int) / (int, int)
column 22 (195: 196): databestgrade2: str / int / int
column 23 (196: 197): databestgrade3: str / int / int
column 24 (197: 198): databestgrade4: str / int / int
column 25 (198: 199): databestgrade5: str / int / int
column 26 (154: 155): databig5_01: str / int / int
column 27 (155: 156): databig5_02: str / int / int
column 28 (156: 157): databig5_03: str / int / int
column 29 (157: 158): databig5_04: str / int / int
column 30 (158: 159): databig5_05: str / int / int
column 31 (159: 160): databig5_06: str / int / int
column 32 (160: 161): databig5_07: str / int / int
column 33 (161: 162): databig5_08: str / int / int
column 34 (162: 163): databig5_09: str / int / int
column 35 (163: 164): databig5_10: str / int / int
column 36 (7: 8): datadiv3filler: str / int / bool
column 37 (189: 190): dataelm_01: str / int / int
column 38 (190: 191): dataelm_02: str / int / int
column 39 (191: 192): dataelm_03: str / int / int
column 40 (192: 193): dataelm_04: str / int / int
column 41 (193: 194): dataelm_05: str / int / int
column 42 (8: 17): dataethnicity: str / int / [bool’s]
column 43 (247: 248): datafeedback: str / int / bool
column 44 (17: 22): datagender: str / int / [bool’s]
column 45 (248: 249): datahighpower: str / int / bool
column 46 (249: 250): datainstructbig5: str / int / bool
column 47 (250: 251): datainstructintrinsic: str / int / bool
column 48 (251: 252): datainstructmli: str / int / bool
column 49 (252: 253): datainstructnfc: str / int / bool
column 50 (22: 23): dataintrinsic_01: str / int / int
column 51 (23: 24): dataintrinsic_02: str / int / int
column 52 (24: 25): dataintrinsic_03: str / int / int
column 53 (25: 26): dataintrinsic_04: str / int / int
column 54 (26: 27): dataintrinsic_05: str / int / int
column 55 (27: 28): dataintrinsic_06: str / int / int
column 56 (28: 29): dataintrinsic_07: str / int / int
column 57 (29: 30): dataintrinsic_08: str / int / int
column 58 (30: 31): dataintrinsic_09: str / int / int
column 59 (31: 32): dataintrinsic_10: str / int / int
column 60 (32: 33): dataintrinsic_11: str / int / int
column 61 (33: 34): dataintrinsic_12: str / int / int
column 62 (34: 35): dataintrinsic_13: str / int / int
column 63 (35: 36): dataintrinsic_14: str / int / int
column 64 (36: 37): dataintrinsic_15: str / int / int
column 65 (131: 134): datakposition: str / int / [bool’s]
column 66 (134: 135): datakratio: str / int / int
column 67 (253: 254): datalowpower: str / int / bool
column 68 (135: 138): datalposition: str / int / [bool’s]
column 69 (138: 139): datalratio: str / int / int
column 70 (37: 46): datamajor: str / int / [bool’s]
column 71 (205: 206): datamcdv1: str / int / int
column 72 (206: 207): datamcdv2: str / int / int
column 73 (207: 208): datamcfiller1: str / int / int
column 74 (208: 209): datamcfiller2: str / int / int
column 75 (209: 210): datamcfiller3: str / int / int
column 76 (210: 211): datamcmost1: str / int / bool
column 77 (211: 212): datamcmost2: str / int / bool
column 78 (212: 213): datamcmost3: str / int / bool
column 79 (213: 214): datamcmost4: str / int / bool
column 80 (214: 215): datamcmost5: str / int / bool
column 81 (215: 216): datamcsome1: str / int / bool
column 82 (216: 217): datamcsome2: str / int / bool
column 83 (217: 218): datamcsome3: str / int / bool
column 84 (218: 219): datamcsome4: str / int / bool
column 85 (219: 220): datamcsome5: str / int / bool
column 86 (46: 54): datamood_01: str / int / [bool’s]
column 87 (54: 62): datamood_02: str / int / [bool’s]
column 88 (62: 63): datanfc_01: str / int / int
column 89 (63: 64): datanfc_02: str / int / int
column 90 (64: 65): datanfc_03: str / int / int
column 91 (65: 66): datanfc_04: str / int / int
column 92 (66: 67): datanfc_05: str / int / int
column 93 (67: 68): datanfc_06: str / int / int
column 94 (139: 142): datanposition: str / int / [bool’s]
column 95 (68: 69): datanratio: str / int / int
column 96 (69: 70): datapate_01: str / int / int
column 97 (70: 71): datapate_02: str / int / int
column 98 (71: 75): datapate_03: str / int / [bool’s]
column 99 (75: 82): datapate_04: str / int / [bool’s]
column 100 (82: 89): datapate_05: str / int / [bool’s]
column 101 (142: 145): datarposition: str / int / [bool’s]
column 102 (145: 146): datarratio: str / int / int
column 103 (174: 175): datasarcasm: str / int / int
column 104 (199: 200): dataselfesteem_01: str / int / int
column 105 (89: 90): datastress_01: str / int / int
column 106 (90: 91): datastress_02: str / int / int
column 107 (91: 92): datastress_03: str / int / int
column 108 (92: 93): datastress_04: str / int / int
column 109 (182: 183): datatempest1: str / int / int
column 110 (183: 184): datatempest2: str / int / int
column 111 (184: 185): datatempest3: str / int / int
column 112 (185: 186): datatempfollowup1: str / int / int
column 113 (186: 187): datatempfollowup2: str / int / int
column 114 (187: 188): datatempfollowup3: str / int / int
column 115 (146: 149): datavposition: str / int / [bool’s]
column 116 (149: 150): datavratio: str / int / int
column 117 (200: 201): dataworstgrade1: str / (int, int) / (int, int)
column 118 (201: 202): dataworstgrade2: str / int / int
column 119 (202: 203): dataworstgrade3: str / int / int
column 120 (203: 204): dataworstgrade4: str / int / int
column 121 (204: 205): dataworstgrade5: str / int / int
column 122 (93: 100): datayear: str / int / [bool’s]
column 123 (254: 255): dataStation: str / int / bool
column 124 (255: 256): dataDate_x: str / int / bool
column 125 (256: 257): dataExperimenter: str / int / bool
column 126 (188: 189): dataTemperatureinlab: str / int / int
column 127 (100: 103): dataOrderofTasks: str / int / [bool’s]
column 128 (175: 178): dataClipboardWeight: str / int / [bool’s]
column 129 (178: 179): dataIIResponse: str / int / int
column 130 (120: 124): dataSRCondition: str / int / [bool’s]
column 131 (124: 128): dataSRMeetingResponse: str / int / [bool’s]
column 132 (128: 129): dataSRConfidenceResponse: str / int / int
column 133 (129: 130): dataSRTFCorrect: str / int / int
column 134 (257: 258): dataNotes: str / int / bool
```

```
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
column 170 (179: 182): dataClipBoardMaterial: str / int / bool
column 171 (103: 104): dataPersistence: str / int / int
```

```
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
```

```
None
None
None
None
None
None
None
None
None
None
None
None
None
column 222 (258: 259): dataMonthComputer: str / int / int
column 223 (259: 260): dataDayComputer: str / int / int
column 224 (260: 261): dataYearComputer: str / int / int
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
None
column 242 (104: 105): dataOpenness: str / int / int
column 243 (105: 106): dataConscientiousness: str / int / int
column 244 (106: 107): dataExtraversion: str / int / int
column 245 (107: 108): dataAgreeableness: str / int / int
column 246 (108: 109): dataNeuroticism: str / int / int
column 247 (109: 110): dataIntrinsic: str / int / int
column 248 (110: 111): dataMood: str / int / int
column 249 (111: 112): dataNFC: str / int / int
column 250 (112: 113): dataReportedAttention: str / int / int
column 251 (113: 114): dataReportedEffort: str / int / int
column 252 (114: 115): dataSelfEsteem: str / int / int
column 253 (115: 116): dataStress: str / int / int
None
column 255 (220: 221): dataMostEndorse: str / int / int
column 256 (221: 222): dataSomeEndorse: str / int / int
column 257 (222: 223): dataCredCond: str / int / bool
column 258 (116: 117): dataGenderfactor: str / int / bool
None
None
None
None
None
None
None
None
None
column 268 (117: 118): dataTempCond: str / int / bool
column 269 (118: 119): dataTargetGender: str / int / bool
column 270 (119: 120): dataArgumentQuality: str / int / int
None
None
None
```