

**GOVERNMENT OF THE RUSSIAN FEDERATION NATIONAL
RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS**

**FACULTY OF ECONOMIC SCIENCES
Educational Programm "Economics"**

Bachelor's Term Paper

On topic "Conjoint Analysis with Bayesian Logistic Regression"

Student of group BEC-162
Tsvigun Akim Olegovich

Academic Supervisor:
Senior Lecturer Demeshev Boris Borisovich

Moscow 2019

Contents

1	Abstract	3
2	Introduction	3
2.1	Basics of Conjoint Analysis	3
2.2	Sample Generation	4
2.3	Evaluation techniques	4
3	Preprocessing data	5
3.1	Creation of features array for aggregate model	5
3.2	Dimensionality reduction	5
4	Model selection	6
4.1	Why logistic regression	6
4.2	Probabilities in conjoint with logistic regression	6
4.3	Logistic regression from scratch	9
4.4	Bayesian logistic regression	11
5	Programming	11
5.1	Aggregate model	11
5.2	Personal models	12
6	Conclusion	12
	References	13
7	Applications	13
7.1	Logistic Regression from scratch	13
7.2	Conjoint Analysis	15

1 Abstract

Marketing is developing at high pace nowadays, and to get a more realistic picture of the customers' preferences companies start using conjoint analysis. It approaches the choice in real life when a person has to select from the offered alternatives. Applying conjoint analysis, producers can find out the most valuable and important characteristics among customers and make their products more market-oriented. However, this type of analysis is well known only among specialists: it is hardly mentioned in data analysis courses or in any books. Furthermore, the most popular statistical programming language, Python, still does not provide any open-source packages for conjoint analysis. That is exactly the reason I decided to devote my term paper to this topic: to facilitate its understanding for further researchers.

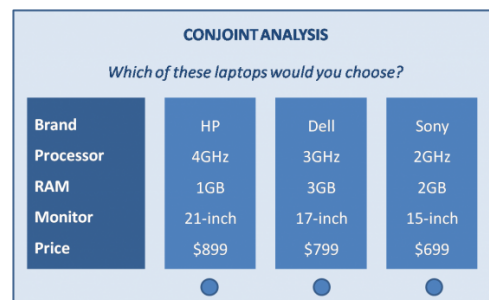
The goal of the paper is to make it clear for any person, who has a solid base in Maths, how to analyze data with conjoint analysis. The code for the work is provided in Python for a range of reasons. Firstly, Python has a very simple syntax, which makes it easy and pleasant to learn. Secondly, Python quickly gains popularity among data scientists and quantative researchers and in my view, it can soon become the principal programming language for artificial intelligence. Moreover, Python possesses an outstanding library Pymc3 for Bayesian Methods, which are thoroughly examined in the paper.

The work is divided into four main blocks: data and its preprocessing, theoretical interpretation of conjoint results, basics of logistic regression and, finally, application of Bayesian logistic regression. There is also the following code in Python for all the parts in the applications.

2 Introduction

2.1 Basics of Conjoint Analysis

Instead of making assessments of the offered goods on a five or ten point scale, a person chooses the most preferable option of the given alternatives. There are various models of conjoint analysis: for instance, instead of choosing the best alternative a person may be asked to range the options or, vice versa, select the worst candidate. To make it even closer to reality, conjoint analysis usually contains a "nothing" option - selecting this means the respondent does not feel like choosing any of the op-



CONJOINT ANALYSIS			
Which of these laptops would you choose?			
Brand	HP	Dell	Sony
Processor	4GHz	3GHz	2GHz
RAM	1GB	3GB	2GB
Monitor	21-inch	17-inch	15-inch
Price	\$899	\$799	\$699

Figure 1.1: Example of the conjoint analysis

tions and thereby decides to skip this type of goods. However, in the paper the sample is generated with only the most preferable alternative chosen and without "nothing" option.

The key advantage of the conjoint analysis over the other assessing methods is that it simulates the real-world purchasing situations that ask respondents to trade one option for another. It lets you gauge the value of a specific offering in the eyes of your target consumers, comparing it with the other alternatives. In addition, latest researches have revealed that people seem to give either the maximum or the minimum evaluation to the product when asked to evaluate goods on an x-point scale. According to the research, when estimating on a five-point scale, only 21 percent of the results are not equal to 1 or 5 on average, which is critically low. As a consequence, samples collected with such techniques of analysis are often unrepresentative.

2.2 Sample Generation

Sampling procedure is not as simple as it may look at first sight. Each of n respondents is given m cards; each card depicts l alternatives; each alternative, in turn, has k features. A person selects the option which appeals to him the most from each card. Therefore, a 4-d table of features and a 2-d matrix of target values are composed. The first dimension in the table represents different respondents and its length equals n . The second dimension of length m depicts the cards. The third, with length equal to l , demonstrates the alternatives. The fourth dimension shows characteristics of the alternatives; its length equals k . Totally, the shape of the features table is equal to $n \times m \times l \times k$.

The matrix of target values contains the number of the selected alternative of each card of each person. Thus, its shape equals $n \times m$. However, as classification models can only predict either 0 or 1, this matrix is then transformed to a 3-d array of shape $n \times m \times l$ where instead of a number from 1 to l each row contains a vector of length t . Coordinate t of this vector represents whether the alternative t from the card was chosen or not (0 if False, 1 if True).

2.3 Evaluation techniques

Being 4-dimensional arrays, samples collected by means of conjoint analysis cannot be treated as ordinary features matrices. The key problem is the value of the target for the alternative (0 or 1) depends on the characteristics of other options, as respondents select the most preferable variant. Thus, in the context of each observation the environment is stochastic: even if we know the true values of the coefficients, we cannot unambiguously find the target value. Such a problem is solved by using recommender systems, which are destined for exactly the same purposes: having a list of options, select the most suitable one. However, as in this paper neither recommendation systems nor neural networks

that would properly process 3-d arrays are used, the sample needs to be either somehow reshaped or modified.

3 Preprocessing data

3.1 Creation of features array for aggregate model

When building an aggregate model for all the respondents, we treat them as a unique market character, customer. As in this case we assume depersonification, all the observations equally contribute to the estimates of coefficients. Thereby, the 4-d array of shape $n \times m \times l \times k$ may be reshaped into a 3-d array with shape $n \cdot m \times l \times k$. In a similar fashion, the 3-d array of goal variable of shape $n \times m \times l$ is transformed into a 2-d matrix, whose shape equals $n \cdot m \times l$.

3.2 Dimensionality reduction

The next step is to make the environment deterministic: transform the data in such a way that the features uniquely determine the output. The method suggested in the paper is called "difference-based" table generation. The idea is as follows: as each time a respondent chooses one option from l alternatives, we create a table of shape $n \cdot m \cdot (l - 1) \times k$, where each object is either a difference of the selected option features vector and features vector of every non-selected alternative, with probability of $p = 0.5$, or vice versa ($=$ multiplying by -1), also with probability of $0.5 (= 1 - p)$. Consequently, target matrix is transformed into a vector of length $n \cdot m \cdot (l - 1)$, equalling to 1 with probability of p and 0 with probability of $1 - p$. Parameter p makes it convenient to balance samples, especially when merging them.

This way we obtain customary features matrix and target vector destined for ordinary two-group classification tasks. The utility of option i is calculated as $w^T x_i$, where w is the vector of coefficients estimates and x_i - features vector of option i . Therefore, as the operation of calculation of product utility is linear, our method meets the condition of transitive relation. Hence, the best option always has the maximum utility value among all the alternatives, so it is always possible to figure out the most preferable alternative, comparing all the variants among each other.

If the aim is to get the coefficients estimates of a personal model, we do not take care of other respondents' answers, thus the features array assumes a shape $m \cdot (l - 1) \times k$ for every person, while the goal variable vector is of length $m \cdot (l - 1)$.

4 Model selection

4.1 Why logistic regression

There are thousands of machine learning models that may be fitted with supervised learning with a simple 2-d array of features and a vector of target values. Many of them can also find very complicated dependences and connections and, as a consequence, predict the data much better than a Bayesian logistic regression. The reasons why it is selected in the paper are as follows. First of all, our goal is to obtain parameters estimates, which will be easy to interpret. If they are not (interpretable), it is hardly possible to make a correct inference based on our data: as we do not possess large amount of data, difficult interrelations may be just a consequence of overfitting. Thereby, we need either a linear model or a very simple neural network with one dense layer (to see the direct effect of every coefficient on the output), which will not be very much distinct from linear models.

Secondly, because the evaluations of coefficients are of more importance for us than the subsequent forecasting, Bayesian approach to parameter estimation seems the most suitable.

At last, using logistic regression along with Gumbel distribution for the errors makes it incredibly simple predict the probability of selecting one or another option. We will ensure it in the next section.

4.2 Probabilities in conjoint with logistic regression

Using logistic regression, calculation of a probability of selecting option i among l candidates is extremely easy. Assume there are l alternatives with k features (each); the residuals are independent and identically distributed (iid) with Gumbel density function:

$$f(x) = \frac{1}{\beta} e^{-(z+e^{-z})}, \quad (1.1)$$

where $z = \frac{x-\mu}{\beta}$, μ — location, (real), β — scale, real and strictly positive. We will use standard Gumbel distribution where $\mu = 0, \beta = 1$. In this case cumulative distribution function takes on a form

$$F(x) = e^{-e^{-x}}, \quad (1.2)$$

while density function looks as

$$f(x) = e^{-(x+e^{-x})}. \quad (1.3)$$

As we use logistic regression, the utility of each alternative linearly depends on the features and weights: $U_i = w^T x_i + \varepsilon_i$ where U_i is a utility of option i , w is a vector of parameters, x_i is a features vector of option i and ε_i is a residual of this variant. We will also label predicted utility $w^T x_i$ as V_i and the choice of the responder as y .

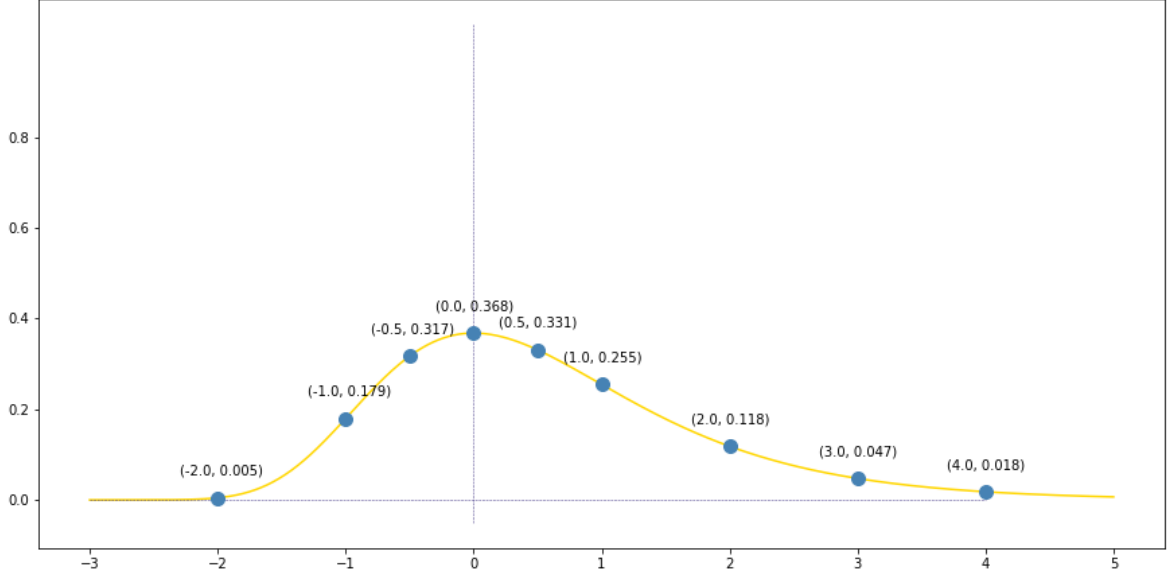


Figure 1.2: Probability density function of standard Gumbel distribution with key points

Probability of that the responder chooses alternative i is as follows:

$$P(y = i) = P(U_i > U_j, \forall j \in l, \forall j \neq i) = P(V_i + \varepsilon_i > V_j + \varepsilon_j, \forall j \neq i) = P(\varepsilon_j < \varepsilon_i + V_i - V_j, j \neq i) \quad (1.4)$$

We have assumed residuals are iid and have a Gumbel distribution. Thus, as cumulative distribution function of a real-valued random variable ξ , evaluated at x , is the probability that ξ takes a value less than or equal to x , we can derive

$$F_{\varepsilon_j}(\varepsilon_i + V_i - V_j) = e^{-e^{-(\varepsilon_i + V_i - V_j)}} \quad (1.5)$$

Taking into account the independence of the residuals (in particular, of ε_i and ε_j), the conditional probability of selecting i with given ε_i is a product of cumulation distribution functions of $\varepsilon_j, \forall j \in l, j \neq i$, evaluated at $\varepsilon_i + V_i - V_j$ - the probability that $U_i \geq U_j, \forall j \neq i$:

$$P(y = i | \varepsilon_i) = \prod_{\forall j \neq i} e^{-e^{-(\varepsilon_i + V_i - V_j)}} \quad (1.6)$$

Consequently,

$$\begin{aligned} P(y = i) &= \int_{-\infty}^{+\infty} P(y = i, \varepsilon_i) d\varepsilon_i = \int_{-\infty}^{+\infty} P(y = i | \varepsilon_i) P(\varepsilon_i) d\varepsilon_i = \\ &= \int_{-\infty}^{+\infty} \prod_{\forall j \neq i} e^{-e^{-(\varepsilon_i + V_i - V_j)}} \cdot P(\varepsilon_i) d\varepsilon_i = \\ &= \int_{-\infty}^{+\infty} \prod_{\forall j \neq i} e^{-e^{-(\varepsilon_i + V_i - V_j)}} \cdot e^{-(\varepsilon_i + e^{-\varepsilon_i})} d\varepsilon_i = \\ &= \int_{-\infty}^{+\infty} \prod_{\forall j \neq i} e^{-e^{-(\varepsilon_i + V_i - V_j)}} \cdot e^{-\varepsilon_i} \cdot e^{-e^{-\varepsilon_i}} d\varepsilon_i \end{aligned} \quad (1.7)$$

We may notice that if $i = j$:

$$e^{-e^{-(\varepsilon_i + V_i - V_j)}} = e^{-e^{-\varepsilon_i}}, \quad (1.8)$$

which is exactly the second factor of the equation (1.7). Thus, the equation may be transformed into:

$$\begin{aligned} P(y = i) &= \int_{-\infty}^{+\infty} \prod_{\forall j} e^{-e^{-(\varepsilon_i + V_i - V_j)}} e^{-\varepsilon_i} d\varepsilon_i = \\ &= \int_{-\infty}^{+\infty} e^{-\sum_{\forall j} e^{-(\varepsilon_i + V_i - V_j)}} \cdot e^{-\varepsilon_i} d\varepsilon_i = \\ &= \int_{-\infty}^{+\infty} e^{-e^{-\varepsilon_i} \cdot \sum_{\forall j} e^{V_j - V_i}} \cdot e^{-\varepsilon_i} d\varepsilon_i \end{aligned} \quad (1.9)$$

To calculate the integral, substitute $e^{-\varepsilon_i} = a$:

$$P(y = i) = \int_{+\infty}^0 e^{-a \cdot \sum_{\forall j} e^{V_j - V_i}} \cdot a \cdot d\varepsilon_i = \int_0^{+\infty} e^{-a \cdot \sum_{\forall j} e^{V_j - V_i}} da \quad (1.10)$$

As $\sum_{\forall j} e^{V_j - V_i}$ is a constant, the integral can be calculated as:

$$\begin{aligned} \int_0^{+\infty} e^{-a \cdot \text{const}} da &= -\frac{e^{-a \cdot \text{const}}}{\text{const}} \Big|_0^{+\infty} = \frac{e^{-a \cdot \sum_{\forall j} e^{V_j - V_i}}}{\sum_{\forall j} e^{V_j - V_i}} \Big|_0^{+\infty} = \\ &= \frac{e^{-0 \cdot \sum_{\forall j} e^{V_j - V_i}}}{\sum_{\forall j} e^{V_j - V_i}} - \frac{e^{-\infty \cdot \sum_{\forall j} e^{V_j - V_i}}}{\sum_{\forall j} e^{V_j - V_i}} = \\ &= \frac{1}{\sum_{\forall j} e^{V_j - V_i}} = \frac{1}{e^{-V_i} \cdot \sum_{\forall j} e^{V_j}} = \frac{e^{V_i}}{\sum_{\forall j} e^{V_j}} \end{aligned} \quad (1.11)$$

As a consequence, the probability of choosing option i among l alternatives is a logistic function:

$$P(y = i) = \frac{e^{V_i}}{\sum_{\forall j} e^{V_j}} \quad (1.12)$$

The fact the choice probability is expressed via a logistic function is a significant advantage of a logit model over others: first of all, because it simplifies the posterior sample generation. For instance, in case of probit model it is necessary to approximate the n -dimensional integral.

However, it is important to treat the assumption of independence of residuals very carefully. Residuals ε_i and ε_j is an unobservable part of utility of options. In other words, it is a difference between real utility and the one modeled by logistic regression. Hence, a vital condition of independence of residuals is a well-specified model, which will include all the required information to make a choice. In that case the unobservable part of utility will be just noise.

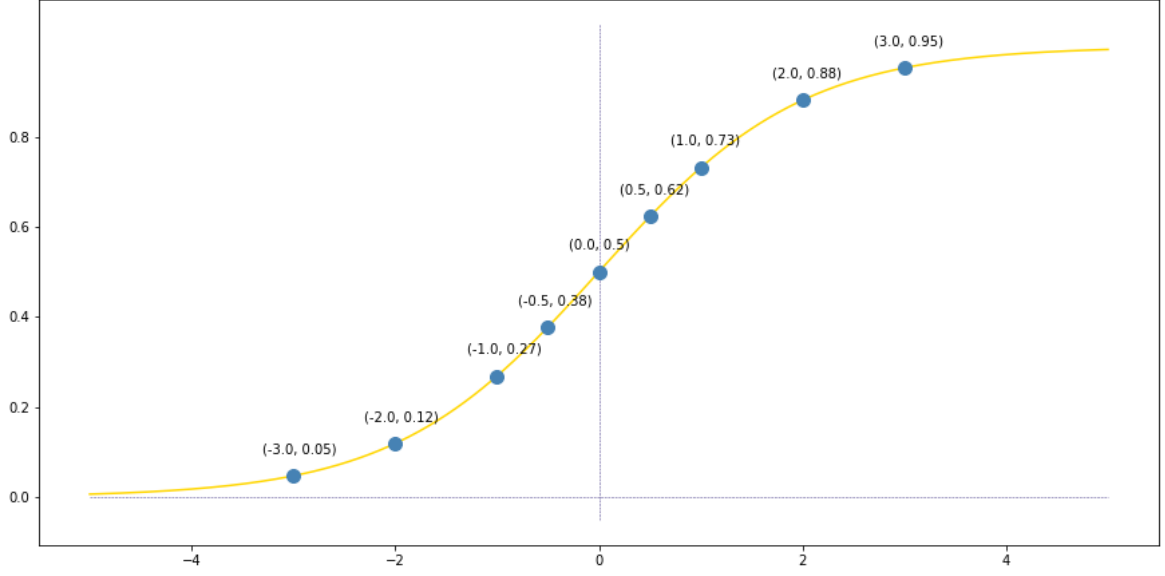


Figure 1.3: Sigmoid graph with key points

4.3 Logistic regression from scratch

¡NB!: In this part we define an operation of applying a function to a vector: $f((x, x_2, \dots, x_n)) = (f(x_1), f(x_2), \dots, f(x_n))$. It signifies the function is applied to each element of the vector. The introduction of this operation is reasonable, because this part is done in Python using the library Numpy, where this operation is defined.

To better understand the operating principle of logistic regression, let's manually deduce how it generates probabilities. Our sample collected by difference-based method is of shape $N \times k$, while the target variable is a binary one (consists of only 0 or 1). Thereby, we need a simple two-group logistic model. The probability for i -th element of sample is calculated via sigmoid function:

$$P(y_i = 1) = \frac{1}{1 + e^{-wx_i}}, \quad (1.13)$$

where y_i is a value of goal variable for object i , w is a vector of coefficients estimates, x_i is a features vector of object i . For convenience, substitute sigmoid function $\frac{1}{1+e^{-wx_i}}$ with $h(wx_i)$. Loss function is obtained by maximizing a likelihood function, which is a product of probabilities: $h(wx_i)$ if $y_i = 1$, and $1 - h(wx_i) = h(-wx_i)$ if $y_i = 0$. Hence, it takes on a form

$$L = \prod_{i=1}^N (h(wx_i)^{y_i=1} \cdot h(-wx_i)^{y_i=0}) \rightarrow \max_w \quad (1.14)$$

Taking a natural logarithm, get

$$\begin{aligned}
l &= \log\left(\prod_{i=1}^N (h(wx_i))^{y_i=1} \cdot (h(-wx_i))^{y_i=0}\right) = \\
&= \sum_{i=1}^N (y_i \log(h(wx_i)) + (1 - y_i) \log(1 - h(wx_i))) \rightarrow \max_w
\end{aligned} \tag{1.15}$$

As loss function must be minimized, we can make an equivalent transition by multiplying our expression by -1 :

$$-\sum_{i=1}^N (y_i \cdot \log(h(wx_i)) + (1 - y_i) \cdot \log(1 - h(wx_i))) \rightarrow \min_w \tag{1.16}$$

We have deduced the loss function of logistic regression. This function is also called log-loss, and the family of such functions - bernoulli or binomial likelihood. Defining operation $f(\text{vector})$ as application of $f()$ to each component of vector , we can rewrite the expression as

$$-y^T \times \log(h(Xw)) + (\vec{1} - y)^T \times \log(\vec{1} - h(Xw)). \tag{1.17}$$

To minimize it, we will use gradient descent method. Its core idea is minimization of some function by iteratively moving in the direction of the steepest descent as defined by the negative of the gradient. Thus, our task is to find the gradient of log loss. First, let's find the derivative of sigmoid function for x - a vector of shape $1 \times k$ (a single observation).

$$\begin{aligned}
\frac{\delta h(xw)}{\delta w} &= \frac{\delta \frac{1}{1+e^{-xw}}}{\delta w} = \frac{\delta (1 + e^{-xw})^{-1}}{\delta w} = \\
&= \frac{\delta (1 + e^{-xw})^{-1}}{\delta e^{-xw}} \cdot \frac{\delta e^{-xw}}{\delta (-xw)} \cdot \frac{\delta (-xw)}{\delta w} = \\
&= -(1 + e^{-xw})^{-2} \cdot e^{-xw} \cdot (-x) dw = \\
&= \frac{1}{1 + e^{-xw}} \cdot \frac{e^{-xw}}{1 + e^{-xw}} \cdot x dw = \\
&= h(xw) \cdot h(-xw) \cdot x dw
\end{aligned} \tag{1.18}$$

The target vector y^T is a constant with respect to w , therefore

$$\nabla - (y^T \times \log(h(Xw))) = -y^T \times \nabla \log(h(Xw)) \tag{1.19}$$

Now we can find the gradient of log loss. It equals

$$\begin{aligned}
\nabla \text{LogLoss} &= -y^T \times \nabla \log(h(Xw)) - (\vec{1} - y)^T \times \\
&\times \nabla \log(\vec{1} - h(Xw)) = -y^T \frac{\delta \log(h(Xw))}{\delta h(Xw)} \times \frac{\delta h(Xw)}{\delta w} - \\
&- (\vec{1} - y)^T \frac{\delta \log(h(-Xw))}{\delta h(-Xw)} \times \frac{\delta h(-Xw)}{\delta w} = \\
&= -y^T h(-Xw) \times \vec{1}^T \times Xdw + \vec{1}^T \times h(Xw) \times \vec{1}^T \times Xdw - \\
&- y^T h(Xw) \times \vec{1}^T \times Xdw = h(Xw)^T \times I_{n \times n} \times \\
&\times Xdw - y^T (h(-Xw) + h(Xw)) \vec{1}^T Xdw = \\
&= (h(Xw)^T - y^T) Xdw = X^T (h(Xw) - y) dw
\end{aligned} \tag{1.20}$$

With the gradient descent, we do m steps, each time subtracting $\gamma \cdot X^T(h(Xw) - y)$ from the vector of weights w , where γ is the learning rate. It stands to reason we are first of all to initialize it. It is usually done either with a zero-vector or using xavier normal initialization. This way we obtain the coefficients estimates for logistic regression. The realization of it can be found in the [Applications \(1\)](#).

4.4 Bayesian logistic regression

Here we will not deepen into the Bayesian methods - the reader can find out more about them here. The peculiarity of Bayesian logistic regression is that its likelihood function is a Bernoulli function. Thereby, the posterior functions for the estimates are calculated via the integral:

$$\begin{aligned}
&\int \frac{P(\theta) \cdot \prod_i [h(\theta x_i)^{y_i} \cdot h(1 - \theta x_i)^{1-y_i}]}{P(X)} \propto \\
&\propto P(\theta) \cdot \prod_i [h(\theta x_i)^{y_i} \cdot h(1 - \theta x_i)^{1-y_i}]
\end{aligned} \tag{1.21}$$

where $h(t)$ is a sigmoid function, and $P(X)$ can be derived out of the integral normalization condition. Parameters values are then sampled from the posterior distribution and this is how we get their estimates.

5 Programming

5.1 Aggregate model

First of all, the data is transformed into the two-dimensional table using difference-based method. With this table we are ready to build common model, which does not take into account personal preferences and considers the respondents as a single agent. To better specify the posterior estimates, we firstly deduce them with simple logistic regression and

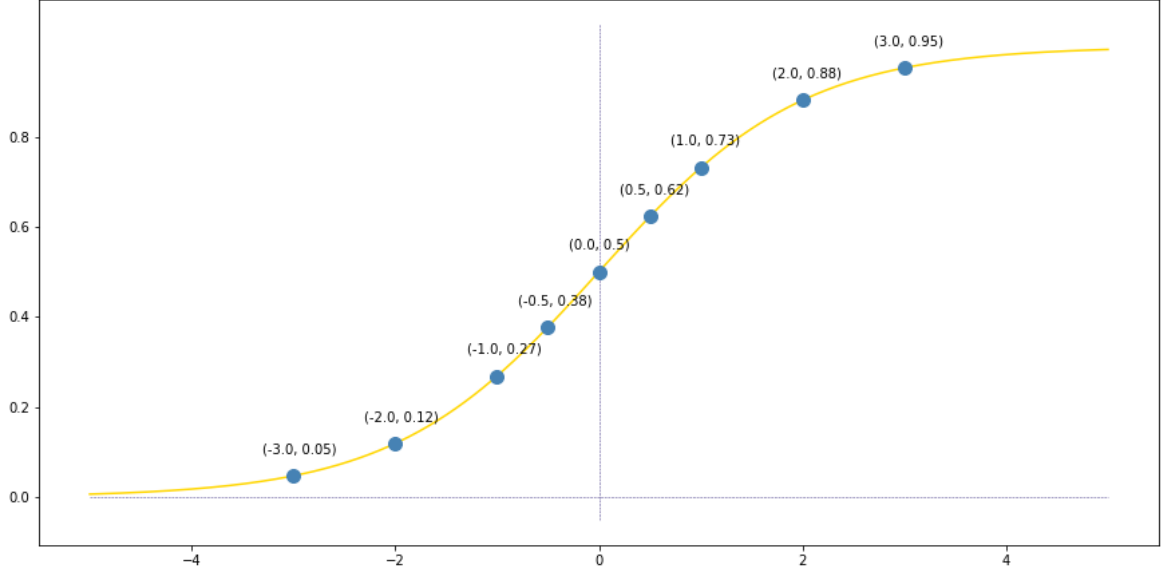


Figure 1.4: Sigmoid graph with key points

then set weak prior distributions with the obtained evaluations as means and large value of variance (to make it weak). We also set Bernoulli likelihood function and use NUTS algorithm to sample the values from the posterior function.

5.2 Personal models

We also need to build personal models for each respondent. As we use Bayes inference, estimates are very close to reality even though we evaluate 11 parameters by 28 observations for each person. We are now aware of the approximate estimates for each person from the common model, thus we may set prior distributions more strictly (reduce the variance). Moreover, we can limit the values of some parameters: for instance, if we are confident the coefficient takes on a positive value, we can use Wald or Log-Normal distribution as a prior for it. Consequently, we get 11-dimensional vector of estimates for each person.

Our next aim is to visualize the output. Here we use t-SNE algorithm, which models a distribution in less dimensional space (on the plain in our case) and approaches it to the true distribution by minimizing Kullback–Leibler divergence. This way we reduce the dimensionality of estimates for each person to 2 and 3 and visualize them.

The code for both models is given in the [Applications \(2\)](#).

6 Conclusion

We have solved a real-life case, where we have unstructured data of conjoint analysis and need to collect valuable information from it. It stands to reason that much more inferences can be made out from the data, however it is not our goal. We have obtained the required

coefficients estimates, built various models and visualized them. More interesting graphs from this data can be found in my repository on github, @aktsvigun The work also allows one to inspect [?] all the conjoint data by applying difference-based method of creating the table and using simple but very precise models: their competence is proved by the graphs from the paper.

7 Applications

7.1 Logistic Regression from scratch

```
import numpy as np

class LogisticRegression:
    def __init__(self, intercept=True, l1=0, l2=0,
                 weights=None, threshold=0.5):
        self.intercept = intercept
        self.l1 = l1
        self.l2 = l2
        self.weights = weights
        self.threshold = threshold

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def __add_intercept(X):
        return np.hstack((np.ones((len(X), 1)), X))

    def __loss(self, w, X, y):
        return -(y.T @ np.log(self.__sigmoid(X@w)) +
                 (1 - y).T @ np.log(self.__sigmoid(-X@w)))

    def __gradient_loss(self, w, X, y):
        return X.T @ (self.__sigmoid(X@w) - y[:, None])

    def fit(self, X, y, learning_rate=0.01,
            num_epoch=10000, verbose=False):
```

```

if not (isinstance(X, np.ndarray) and isinstance(
    y, np.ndarray)):
    X, y = np.array(X), np.array(y)
if len(X.shape) != 2:
    X = X.reshape(1, -1)

if self.intercept:
    X = LogisticRegression.__add_intercept(X)

self.weights = np.zeros((X.shape[1], 1)) if isinstance(
    self.weights, type(None)) else self.weights

for epoch in range(1, num_epoch + 1):
    self.weights -= learning_rate * self.__gradient_loss(
        self.weights, X, y)
    if verbose and epoch % 1e+4 == 0:
        print(f'Epoch: {epoch}; loss: \
            {self.__loss(self.weights, X, y)}; \
            weights:{self.weights}.')

def predict_proba(self, X):

    if not isinstance(X, np.ndarray):
        X = np.array(X)
    if len(X.shape) != 2:
        X = X.reshape(1, -1)

    if self.intercept:
        X = LogisticRegression.__add_intercept(X)
    return np.ravel(self.__sigmoid(X@self.weights))

def predict(self, X):
    return 1*(self.predict_proba(X) >= self.threshold)

def get_weights(self):
    return np.ravel(self.weights)

```

7.2 Conjoint Analysis

```
import numpy as np
import pandas as pd
import pyreadstat
import pymc3 as pm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.linear_model import LogisticRegression
from sklearn.manifold import TSNE
from tqdm import tqdm_notebook as tqdm
from multiprocessing import Process, current_process, Manager

def plot_traces(traces, retain=1000):
    """
    Convenience function: plot traces with overlaid means and values
    """

    ax = pm.traceplot(traces[-retain:],
                      figsize=(12, len(traces.varnames)*1.5),
                      lines={k: v['mean'] for k, v in \
                             pm.summary(traces[-retain:]).iterrows()})

    for i, mn in enumerate(
        pm.summary(traces[-retain:])['mean']):

        ax[i,0].annotate('{:.2f}'.format(mn), xy=(mn,0),
                        xycoords='data', xytext=(5,10),
                        textcoords='offset points', rotation=90,
                        va='bottom', fontsize='large',
                        color='#AA0022')

def get_lr_estimates(df, target, random_state=0):
    """
    Get weights estimates using logistic regression.

    arguments:
        df: np.array, matrix of features.
```

```

        target: list or np.array, vector of target values.

    returns:
        estimates for weights using logistic regression.
    """
    lr = LogisticRegression(random_state=random_state)
    lr.fit(df, target)
    return np.concatenate((lr.intercept_, lr.coef_[0]))

def diff_model(df, features, random_state=0):

    """
    Builds a DataFrame with each row as a difference of
    the winning alternative and another one or vice versa.

    arguments:
        df: pd.DataFrame, which to parse.
        features: list or np.array or tuple, characteristics
        of each alternative.
        random_state: int, default=0, random state
        for np.random.seed.

    returns:
        df with difference as each row. Length ==
        num_people * num_cards * (num_alternatives - 1)

    """
    df_diff = pd.DataFrame(columns=np.append(features, 'target'))
    # Fix random seed
    np.random.seed(random_state)
    # For each person
    for num_person in range(len(df)):
        # For each card
        for num_test in range(1, 8):
            # Winning alternative number
            win_number = df.loc[num_person, f'T{num_test}_select']
            # Winning card's characteristics

```



```

char_win = df.loc[num_person,
                  ['T{0}_C{1}_{2}'.format(
                      num_test, win_number, features[i]) \
                      for i in range(10)]] .values
# For each of lose alternatives
for num_obj in np.delete(range(1, 6), win_number-1):
    char_lose = df.loc[num_person,
                      ['T{0}_C{1}_{2}'.format(
                          num_test, num_obj, features[i]) \
                          for i in range(10)]] .values
    # What to add to target, 0 or 1
    win_first = np.random.randint(2)
    if win_first:
        df_diff.loc[len(df_diff)] = np.append(
            char_win - char_lose, win_first)
    else:
        df_diff.loc[len(df_diff)] = np.append(
            char_lose - char_win, win_first)
return df_diff

def personal_model(df, features, model='lr', priors=None,
                  df_slice=None, estimates=None, random_state=0):

    """

    Builds a DataFrame with each row as a difference of
    the winning alternative and another one or vice versa

    arguments:
        df: pd.DataFrame, which to parse.
        features: list or np.array, characteristics of
            each alternative.
        model: 'lr' or 'bayes', default='lr', which model to use.
        priors: dict, default=None, dict of prior values for
            estimates if model='bayes'.
        df_slice: tuple or int, default=None, range(df_slice) is
            the slice of the df over which to iterate.
        estimates: list, default=None, a list if needed to

```

```

        append the results.
        random_state: int, default=0, random state
        for np.random.seed.

    returns:
        df with difference as each row. Length ==
        num_people * num_cards * (num_alternatives - 1)

    """
    print(f'Process {current_process().name} started!')
    np.random.seed(random_state)
    if estimates == None:
        estimates = []
    if not df_slice:
        df_slice = (len(df))
    for num_person in range(*df_slice):
        df_person = pd.DataFrame(columns=np.append(features,
                                                    'target'))

        for num_test in range(1, 8):
            # For each object except for the winning:
            win_number = df.loc[num_person,
                                f'T{num_test}_select']
            char_win = df.loc[num_person,
                               ['T{0}_C{1}_{2}'.format(
                                   num_test, win_number, features[i]) \
                                   for i in range(10)]] .values
            for num_obj in np.delete(range(1, 6), win_number-1):
                char_lose = df.loc[num_person,
                                    ['T{0}_C{1}_{2}'.format(
                                        num_test, num_obj, features[i]) \
                                        for i in range(10)]] .values
            win_first = np.random.randint(2)
            if win_first:
                df_person.loc[len(df_person)] = np.append(
                    char_win - char_lose, win_first)
            else:
                df_person.loc[len(df_person)] = np.append(
                    char_lose - char_win, win_first)

```

```

df_person[['Payment', 'Personalization', 'Price']] *= -1

if model=='bayes':
    with pm.Model() as logistic_model:

        pm.glm.GLM.from_formula('target ~ {0}'.format(
            ' '.join(list(map(lambda x: str(x)+' '+'+',
df_person.columns[:-1]))))[:-2]),
            data=df_person,
            family=pm.glm.families.Binomial(), priors=priors)
        trace_logistic_model = pm.sample(2000, step=pm.NUTS(),
            chains=1, tune=1000)
        estimates.append(np.mean(list(map(lambda x: list(
            x.values()), trace_logistic_model)), axis=0))

else:
    estimates.append(get_lr_estimates(df_person.values[:, :-1],
        df_person.target.values))

print(f'Process {current_process().name} finished!')
return np.array(estimates)

def visualize_estimates(estimates, annotate=True, use_features=None,
    three_dim=True, use_tsne=True, perplexity=6,
    features=None, enlight=None, savefig=False):

    """

    Visualizes the vectors of any dimensionality in 2-d or
    3-d using t-SNE if needed

    arguments:
        estimates: list or np.array, what to visualize.
        annotate: boolean, default=True, whether to annotate
        the points on the graph.
        use_features: list or np.array or list of str or None,
        default=None, which features to visualize. If array
        of ints, responding columns from estimates are selected.

```

If list of str, responding values for features are selected from estimates. If None, all the estimates array is selected.

three_dim: boolean, default=True, whether to visualize in 3-d when using t-SNE.

use_tsne: boolean, default=True, whether to use t-SNE algorithm to visualize. ;Important! if False, features dimensionality must be equal to 2 or 3, either ValueError.

perplexity: int, default=6, which perplexity to use if t-SNE is chosen.

features: list or np.array, default=None, which features to visualize if use_features consists of str.

enlight: int or None, default=None, the number of feature (starting with 0) by which to enlight the points.

The procedure is as follows: the values are divided into 10 equal groups, determined by quantiles 10-90%.

savefig: boolean, default=False, whether to save graph or not.

returns:

None; depicts the graph/s into the environment.

```

"""
if isinstance(enlight, int):
    color = estimates[:, enlight]
else:
    color = 'Aquamarine'
if not use_features:
    tsne = TSNE(2, perplexity)
    points = tsne.fit_transform(estimates)
    if annotate:
        plt.figure(figsize=(25, 25))
        ax = plt.gca()
        ax.set_facecolor('AliceBlue')
        plt.scatter(points[:, 0], points[:, 1], c=color,
                    cmap='viridis', s=100)
    for n in range(len(points)):

```

```

        plt.annotate(f'{n}', xy=points[n, :],
                     textcoords='data')

    else:
        plt.figure(figsize=(10, 5))
        ax = plt.gca()
        ax.set_facecolor('AliceBlue')
        plt.scatter(points[:, 0], points[:, 1], c=color,
                    cmap='viridis', s=100)

    if type(enlight) == int:
        plt.colorbar()
    if savefig:
        plt.savefig('estimates_2_dim.png')

    if three_dim:
        tsne = TSNE(3, perplexity)
        points = tsne.fit_transform(estimates)
        fig = plt.figure(figsize=(20, 10))
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(points[:, 0], points[:, 1], points[:, 2]);
        if savefig:
            plt.savefig('estimates_3_dim.png')
    else:
        if type(use_features[0])=='str':
            use_features = [np.where(features==i)[0][0] \
                           for i in use_features]
        if use_tsne:
            tsne = TSNE(2, perplexity)
            points = tsne.fit_transform(estimates[:, use_features])
            if annotate:
                plt.figure(figsize=(25, 25))
                ax = plt.gca()
                ax.set_facecolor('AliceBlue')
                plt.scatter(points[:, 0], points[:, 1], c=color,
                            cmap='viridis', s=100)
                for n in range(len(points)):
                    plt.annotate(f'{n}', xy=points[n, :],
                               textcoords='data')

```

```

else:
    plt.figure(figsize=(10, 5))
    ax = plt.gca()
    ax.set_facecolor('AliceBlue')
    plt.scatter(points[:, 0], points[:, 1], c=color,
                cmap='viridis', s=100)

if type(enlight) == int:
    plt.colorbar()
if savefig:
    plt.savefig('estimates_2_dim.png')

if three_dim:
    tsne = TSNE(3, perplexity)
    points = tsne.fit_transform(estimates[:, use_features])
    fig = plt.figure(figsize=(20, 10))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(points[:, 0], points[:, 1], points[:, 2]);
    if savefig:
        plt.savefig('estimates_3_dim.png')
else:
    if len(use_features)==2:
        points = estimates[:, use_features]
        if annotate:
            plt.figure(figsize=(25, 25))
            ax = plt.gca()
            ax.set_facecolor('AliceBlue')
            plt.scatter(points[:, 0], points[:, 1], c=color,
                        cmap='viridis', s=100)
            for n in range(len(points)):
                plt.annotate(f'{n}', xy=points[n, :],
                            textcoords='data')
        else:
            plt.figure(figsize=(10, 5))
            ax = plt.gca()
            ax.set_facecolor('AliceBlue')
            plt.scatter(points[:, 0], points[:, 1],
                        c=color, cmap='viridis', s=100)

```

```

        if type(enlight) == int:
            plt.colorbar()
        if savefig:
            plt.savefig('estimates_2_dim.png')

    elif len(use_features)==3:
        points = estimates[:, use_features]
        fig = plt.figure(figsize=(20, 10))
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(points[:, 0], points[:, 1], points[:, 2]);
        if savefig:
            plt.savefig('estimates_3_dim.png')
    else:
        raise ValueError('Please, set 2 or 3 features \
to visualize or use t-SNE.')

def parallel_function(function, args, use_len=True):
    manager = Manager()
    procs = []
    res_1 = manager.list()
    res_2 = manager.list()
    res_3 = manager.list()
    res_4 = manager.list()
    res_5 = manager.list()
    res_6 = manager.list()
    res_7 = manager.list()
    res_8 = manager.list()

    if use_len:
        main_args = args[:-1]
        length = args[-1]

        proc_1 = Process(target=function,
                        args=np.append(main_args,
                                      [(0, length//8),
                                      res_1]))

        procs.append(proc_1)

```

```

proc_2 = Process(target=function,
                  args=np.append(main_args,
                                [(length//8, length*2//8),
                                 res_2]))
procs.append(proc_2)

proc_3 = Process(target=function,
                  args=np.append(main_args,
                                [(length*2//8, length*3//8),
                                 res_3]))
procs.append(proc_3)

proc_4 = Process(target=function,
                  args=np.append(main_args,
                                [(length*3//8, length*4//8),
                                 res_4]))
procs.append(proc_4)

proc_5 = Process(target=function,
                  args=np.append(main_args,
                                [(length*4//8, length*5//8),
                                 res_5]))
procs.append(proc_5)

proc_6 = Process(target=function,
                  args=np.append(main_args,
                                [(length*5//8, length*6//8),
                                 res_6]))
procs.append(proc_6)

proc_7 = Process(target=function,
                  args=np.append(main_args,
                                [(length*6//8, length*7//8),
                                 res_7]))
procs.append(proc_7)

proc_8 = Process(target=function,
                  args=np.append(main_args,

```



```

                                [(length*7//8, length),
                                res_8]))

    procs.append(proc_8)

    for proc in procs:
        proc.start()
    for proc in procs:
        proc.join()

    estimates = np.concatenate((res_1, res_2, res_3, res_4,
                                res_5, res_6, res_7, res_8))

    return estimates

if __name__ == '__main__':
    df, meta = pyreadstat.read_sav('conjoint_host_sim_dummy.sav')
    for n in range(1, 8):
        df[f'T{n}_select'] = df[f'T{n}_select'].astype(int)
    features = np.delete(np.unique(list(
        map(lambda x: x[x.rindex('_')+1:],
            df.columns[2:]))), -1)
    df_diff = diff_model(df, features)
    with pm.Model() as logistic_model:
        pm.glm.GLM.from_formula('target ~ {0}'.format(' '.join(
            list(map(lambda x: str(x)+' '+'+',
                df_diff.columns[:-1]))[:-2])),
            data=df_diff, family=pm.glm.families.Binomial())
        trace_logistic_model = pm.sample(2000, step=pm.NUTS(),
            chains=1, tune=1000)

    plot_traces(trace_logistic_model);
    print(pm.summary(trace_logistic_model))

    priors = dict()
    priors['Intercept'] = pm.Laplace.dist(0, 0.2)
    priors['Gigabytes'] = pm.HalfStudentT.dist(20, 0.18)
    priors['Hostprovider1'] = pm.HalfStudentT.dist(20, 0.8)
    priors['Hostprovider2'] = pm.Lognormal.dist(-1.3, 1)
    priors['Hostprovider3'] = pm.Lognormal.dist(-2, 1)
    priors['Minutes'] = pm.Lognormal.dist(-1.3, 1.5)

```

```

priors['Payment'] = pm.Lognormal.dist(-3.2, 1)
priors['Personalization'] = pm.Lognormal.dist(-2.5, 1)
priors['Price'] = pm.Wald.dist(0.5, 0.5)
priors['Quantitysim2'] = pm.HalfStudentT.dist(20, 0.2)
priors['Quantitysim3'] = pm.Wald.dist(0.2, 0.5)

names_estimates = parallel_function(personal_model,
                                     (df, features, 'bayes',
                                      priors, len(df)))
estimates = np.hstack((np.expand_dims(names_estimates[:, 0], -1),
                       names_estimates[:, 11:]))
visualize_estimates(estimates, savefig=True, features=features,
                    enlight=1, three_dim=False)

```