Learning with Dependent Data

by

Yurunyun Wang

Supervised by Dr. Lam Ho

Submitted in partial fulfillment of the requirements for the degree

of Bachelor of Science: Combined Honours in Statistics

and Computer Science

at

Dalhousie University

Department of Mathematics and Statistics

Halifax, Nova Scotia

April 2021

Contents

1 Acknowlegment

I would like to express my sincerest gratitude to the people who have contributed to the writing of this honor thesis.

I would like to acknowledge the guidance of my thesis supervisor Dr. Lam Ho, who has a great attitude and broad knowledge. Without his dedication, clear guidance, and persistent help this thesis would not have been possible. From the topic selection, data generation coding, to the thesis writing, Dr. Lam Ho gave me many valuable and kindly suggestions, numerous patient and detailed guidance in the busy schedule to guide the code and the thesis, which gave me great support and help at the last moment of my undergraduate career.

In addition, a thank to Professor Keith Thompson for providing me a detailed introduction to possible supervisors in the department of Mathematics and Statistics and kindly recommended Dr. Lam Ho to me.

2 Abstract

This thesis describes research on the differences between learning with independent data and learning with dependent data. We first introduce the background knowledge of Multivariate Normal Distribution and Matrix Normal Distribution, which are the basis of generating data. Next, we introduce 6 commonly used machine learning models that are involved in our project to provide the predicted value, meanwhile we also explain the process of generating both datasets. Then we collect accuracy results from 6 models by input different parameters. We close by discussing the conclusions made by observing the charts of accuracy trend with different dataset types and parameters, such as independent and dependent data, number of observations, number of dimensions, and correlation coefficient.

3 Introduction

3.1 Background

3.1.1 Multivariate Normal Distribution

In this project, we will involve a dataset composed of multi-dimensional variables. So, we may refer to the background knowledge of Multivariate Normal Distribution and Matrix Normal Distribution. The Multivariate Normal Distribution is the base of Matrix Normal Distribution. It is also known as Multivariate Gaussian Distribution or Joint Normal Distribution. Compared to Normal Distribution, it is a generalization to a higher dimension. To define it clearly, suppose there is a d-dimensional random variable denoted as $X_1, \cdots, X_d$, if every linear combination of d components of this random variable is normal distributed, then we can say this random variable follows the Multivariate Normal Distribution, which can be denoted as $X \sim \mathcal{N}_d(\mu, \Sigma)$. The mathematical definition is shown below [15]:

Definition 1 (Multivariate Normal Distribution) *Given a random vector $X = [X_1, \cdots, X_d]^T$ has a multivariate normal distribution. Then we can say X follows a multivariate normal distribution if it meets one of following conditions.*

1. Every linear combineation is denoted as $Y = a_1X_1 + a_2X_2 + \cdots + a_dX_d$, Y's components is normally distributed for any constant vector $a \in \mathbb{R}^d$.

2. There exists mean vector μ with d dimentions and a symmetric, positive semidefinite covariance matrix Σ with a size of $d \times d$, such that the characteristic function of X follows conditions below:

$$\varphi_{X(u)} = exp(iu^T\mu - \frac{1}{2}u^T\Sigma u)$$

Multivariate Normal Distribution has two parameters, mean vector with a size of

$d \times 1$ and covariance matrix with a size of $d \times d$, which is denoted respectively as:

$$\mu = \mathbf{E}\left[X\right] = \left(\mathbf{E}\left[X_1\right], \mathbf{E}\left[X_2\right], \cdots, \mathbf{E}\left[X_d\right]\right)^T$$

$$\Sigma_{i,j} = \mathbf{E}\left[(X_i - \mu_i)(X_j - \mu_j)\right] = Cov\left[X_i, X_j\right]$$

, where $1 \geq i$ and $j \leq d$.

3.1.2 Block Matrix

If a matrix can be divided into blocks or submatrices with an arbitrary size, then this matrix is defined as a block matrix or a partitioned matrix. Intuitively, the block matrix can be separated by several vertical lines and horizontal lines. For instance, the original matrix $A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$, the block matrix is: $A = \begin{bmatrix} A_1 1 & A_1 2 \\ A_2 1 & A_2 2 \end{bmatrix}$.

This is to say, it can be broken into four blocks, which are:

$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$, $A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}$, $A_{21} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix}$, $A_{22} = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}$.

3.1.3 Kronecker Product

In mathematics, the Kronecker Product is an operation to calculate the product of two matrices with arbitrary size. Below is the mathematical definition of the Kronecker Project:

Definition 2 (Kronecker Product) *Given that A matrix has a size of $m \times n$ and B matrix has a size of $p \times q$. Then the Kronecker product of A and B is denoted as $A \otimes B$ with a size of $pm \times qn$:*

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}$$

3.1.4 Matrix Normal Distribution

The Matrix Normal Distribution is utilized to generate dependent multi-dimensional data in this project. In probability and statistics, the Matrix Normal Distribution is defined as the generalization of Multivariate Normal Distribution to applied for matrix-valued random variables. The mathematical definition is shown below:

Definition 3 (Matrix Normal Distribution)

$$X \sim \mathcal{MN}_{m\times d}(M, U, V)$$

if and only if

$$vec(X) \sim \mathcal{N}_{md}(vec(M), V \otimes U)$$

, where $\otimes$ represents the process of Kronecker product introduced previously in section 3.1.3: Kronecker Product.

3.2 Contex

The remainder of this thesis will be outlined as follows. In section 4: Models and Method, the related 6 models will be introduced. The process of generating the dataset is explained in section 5: Data in detail. In section 6: Results, the accuracy results of models by using both independent data and dependent data will be listed in the form of tables. The discussion part and conclusion part can be found respectively in section 7: Discussion and section 8: Conclusion. In section 10: Appendix, the Jupiter notebook files will be attached in the appendix.

4 Models and Method

In this section, we provide a brief introduction of 6 machine learning models utilized in our program. In order to unify the symbol and size of each parameter, we denote X as the d-dimension predictors and denote Y as the outcome, where X has a size of $n \times d$, Y only equals to 0 or 1 and has the same size as $1 \times n$. For dataset X, the number of dimensions is denoted as d.

4.1 Logistic Regression Model

Definition 4 (Sigmoid Function) *[9] Sigmoid function a transformation function used to convert Linear Regression to Logistic regression. The Sigmoid function is defined as*

$$S(x) = \frac{1}{(1+e^x)} = \frac{e^x}{1+e^x}$$

Where S is the sigmoind function , X is the data defined in the beginning of this section. Any real-valued input of sigmoid function can have a result of a value in the range [0,1].

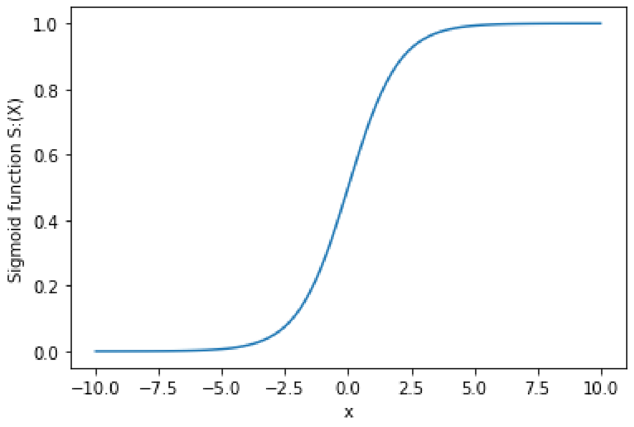
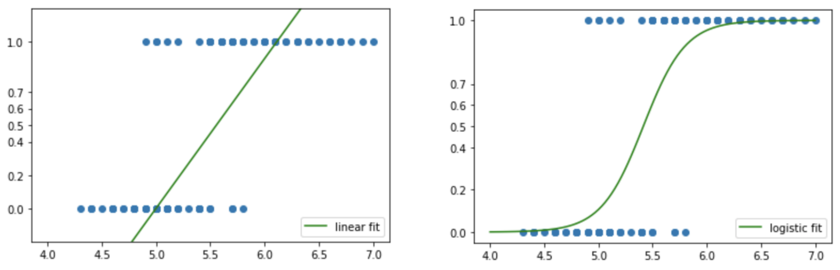


Figure 1: The charts of Sigmoid Function.

Logistic regression is also widely known as logit regression, which is one of the common models used in supervised learning. The logistic regression can also be regarded as a generalized linear model (GLM), which works well for categorical response variable with 2 levels. However, Logistic Regression has a more complex cost function called 'Sigmoid Function' [3].



(a) a (b) b

Figure 2: [a] shows a graph of a linear regression function. Noted that predicted y can exceed 0 and 1; [b] shows a graph of a logistic regression function. Noted that the predicted y only lies between 0 and 1.

According the graph of sigmoid function, it can be seen that the sigmoid function has a bell-curve and its value ranges from 0 to 1. Since the probability of anything only existed in the range of $[0, 1]$, there is no doubt the sigmoid function is a perfect choice for logistic regression [4].

Let the X_i denote the data, where i belongs to the range $0 \leq i \leq n$, and β_i represents the weight of every data. Then the probability of data is defined as

$$p(X_i) = \beta_0 + \beta_1 X_i + \beta_2 X_2 + \cdots + \beta_i X_i + \cdots + \beta_n X_n, \quad 0 \leq i \leq n$$

To avoid the situation that the predicted probability is out of the range $[0, 1]$, the sigmoid function can be used to limit the outcomes. By combining the probability function and sigmoid function, then we obtain the logistic sigmoid function as

$$F(x) = \frac{1}{1 + e^{-x}}$$

So, the cost function can be expressed as

$$S(X_i) = \frac{exp(\beta_0 + \beta_1 X_i + \beta_2 X_2 + \cdots + \beta_i X_i + \cdots + \beta_n X_n)}{1 + exp(\beta_0 + \beta_1 X_i + \beta_2 X_2 + \cdots + \beta_i X_i + \cdot + \beta_n X_n)}, \quad 0 \le i \le n$$

This transformation process is not only used in the logistic model we created for training and testing, but also used in the generating predict values for data. The details are explained in the section 5.3: Generating predict values for data matrix X.

Definition 5 (Cost Function) *A cost function is defined as a measure of how wrong the model performs in terms of its estimation and the true relationship between data X and outcome Y. Mathematically, the cost function for linear regression is expressed as*

$$J = \frac{1}{2}\sum_{i=1}^{n}\big(S(X_i) - y_i\big)^2$$

where n denotes the number of data X, Y_i is the outcome.

After the above transformation, we have used the logistic model to predict the outcome based on the given data X. At this time, a cost function can typically help find a more accurate model by minimizing the value of the cost function. As mentioned in definition 5 the cost function measures how different the predicted value and the actual value are. However, the cost function for linear regression doesn't work when it comes to logistic regression [5]. The reason for this is the graph of logistic regression cost function may have many local minimums so that it is hard to determine which one is the global minimums. That is to say, the cost function for logistic regression is a non-convex function [3].

As a result, the cost function of logistic regression can be expressed mathematically as following

$$Cost(S(X_i), Y_i) = \begin{cases} -log(S(X_i)) & Y_i = 1 \\ -log(1 - S(X_i)) & Y_i = 0 \end{cases}$$

The above cost functions can be combined as a single function like this:

$$J = -\frac{1}{n}\sum[Y_i log(S(X_i)) + (1 - Y_i log(1 - S(X_i)))]$$

4.2 K-Nearest Neighbor Model (KNN algorithm)

K - Nearest Neighbor is an algorithm used in classification problems. During processing the dataset, it continuously stores new added data points from dataset and classifies it based on the votes of surrounding nearest neighbors. The k in the name of this algorithm represents the number of nearest neighbors. When a new data point is added, this algorithm will choose the k nearest neighbors of this data point and figure out the cluster with more votes from k nearest neighbors [13].

The most critical point of implementing the KNN algorithm is to choose a proper number of neighbors (k value). The small k may be noisy and lead to a critical effect on the classification result. On the other hand, even though a large k brings a smoother decision boundary, the classification result is still dramatically impacted due to a lower variance as well as a higher bias [13]. Correspondingly, a larger k value means a more expensive computation because the distance between k nearest neighbors and the newly added data point will be recalculated in every step. In terms of the KNN algorithm, there is no fixed way to figure out the k value. Three commonly used method are listed below

1. The most general way to decide a k value is to let $k = \sqrt{n}$, where n is the total amount of data point in training dataset.

2. The proper k value can be obtained by trying different odd numbers. Because the classification result of KNN model relies on the cluster with more votes, then the odd number can avoid the conflict that two clusters hold the same number of votes from nearest neighbors.

3. To obtain a more precise k value, cross-validation is a better way to approach it. The cross-validation process usually starts with selecting a cross-validation dataset from training data with a small portion around 20%. Then, comparing the predicting

performance while using different possible k values on the cross-validation dataset. Therefore, the k value with the best evaluation on cross-validation dataset should be used in the KNN model.

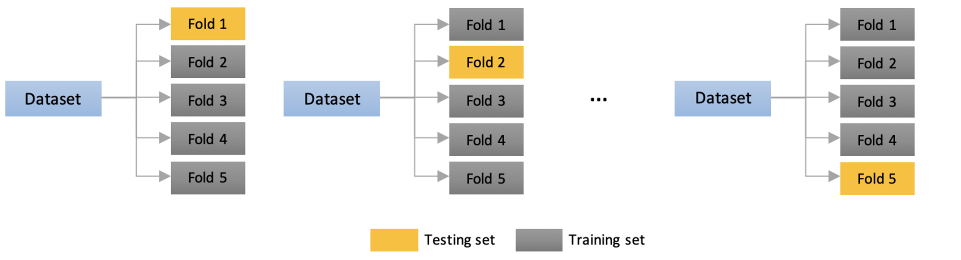


Figure 3: The K-Fold cross-validation process.

When it comes to how KNN algorithm works, it classifies a new data point based on the length of line segment between data points. In this case, we should consider the Euclidean distance with a higher dimension d, the formula below is the general Euclidean distance formula between data point $m = (m_1, m_2, \cdots, m_d)$ and $n = (n_1, n_2, \cdots, n_d)$, where $0 \le i \le d$.

$$d(m,n) = \sqrt{(m_1 - n_1)^2 + (m_2 - n_2)^2 + \cdots + (m_i - n_i)^2 + \cdots + (m_d - n_d)^2}$$

For instance, there are 8 data points, and we need to decide which cluster the data point (4,5) belongs to.

There is a simple example to help understand the procedure of how to decide the cluser of a newly added data point.Assume that $k = 5$. The steps to implement KNN algorithm are as follows:

1. List all the data into a table so that we can have a clear view of all data. The original data points can be seen from above figure.

2. Except for the newly added data point or the target data point, calculate the Euclidean distance between the target data point and all remaining stored data points.

Point	Cluster
(1,5)	0
(1,7)	1
(2,2)	0
(2,7)	0
(5,1)	1
(6,6)	1
(8,6)	1
(4,5) Newly added	?

Figure 4: The original data points and the newly added data point.

Point	Cluster	Distance to point (4,5)
(1,5)	0	3
(1,7)	1	3.61
(2,2)	0	3.61
(2,7)	0	2.83
(5,1)	1	4.12
(6,6)	1	2.24
(8,6)	1	4.12
(4,5) Newly added	?	-

Figure 5: The Euclidean distance between new data point and the rest of points.

3. Rank all the distances from small values to large values and pick up the data point with first k ranks. In this case, the data points with the top 5 ranks are picked up.

Point	Cluster	Distance to point (4,5)	Rank
(1,5)	0	3	3
(1,7)	1	3.61	4 or5
(2,2)	0	3.61	4 or 5
(2,7)	0	2.83	2
(5,1)	1	4.12	6
(6,6)	1	2.24	1
(8,6)	1	4.12	7
(4,5) Newly added	?	-	-

Figure 6: Rank all the distance and find the top 5.

4. Make the decision that the target data point belongs to the cluster with more votes among all 5 nearest neighbors.

Point	Cluster	Distance to point (4,5)	Rank
(1,5)	0	3	3
(1,7)	1	3.61	4 or5
(2,2)	0	3.61	4 or 5
(2,7)	0	2.83	2
(5,1)	1	4.12	6
(6,6)	1	2.24	1
(8,6)	1	4.12	7
(4,5) Newly added	0	-	-

Figure 7: Make a decision based on the cluster with more votes.

4.3 Support Vector Machine Model (SVM)

Similar to Logistic Regression, Support Vector Machine (SVM) also belongs to the common supervised learning models. It is suitable to solve classification problems especially when there are only two groups of data. In this project, we have only two outcomes that are 0 and 1. So, we regard 0 and 1 as the two tags which will be analyzed later in SVM model. Additionally, our data has d dimension, which can be regarded as d features [12].

To explain the process of SVM processing data, all data should be treated as data points. Each data point has d features, what's more, all data points are separated into two groups with tag 0 or tag 1 by a hyperplane. The hyperplane is also called the decision boundary, in other words, the data points laid out on one side of hyperplane have the same tag. To figure out the best hyperplane for classification, we need to maximize the margins from both sides (tags). Every time a new data point is added, the hyperplane is updated by calculating a larger margin until the hyperplane reaches an optimal state.

The hyperplane often works when the data is linear, nevertheless, the hyperplane can hardly classify the data points. In this case, we need to add more dimensions to make sure that the data is linearly connected, meanwhile, put all data in a d-dimensional space. At this time, SVM can obtain an optimal hyperplane and map it back to a lower dimension [12]. In this way, the classification tasks can be finished.

4.4 Random Forest Model

The Random Forest model is one of the commonly used models in the field of Machine Learning and Data Mining. This model performs well when dealing with classification problems. As you can know from the name of the Random Forest Model, it consists of a bunch of decision trees created on the basis of generated random samples [1]. Meanwhile, those decision trees also follow different rules to split the nodes. For those random samples, they are generated by using the Bagging Method, which is a combination of the bootstrap method and the aggregating method.

When it comes to implementing the building block of Random Forest, the decision trees, sometimes we encounter the situation that the training set is overfitting. However, the Random Decision Forests correct this issue as the Bagging method help reduces the variance and avoids overfitting. Following is a brief introduction of the Bagging method.

The Bagging method is also called bagging or bootstrap aggregating. It aims to improve the stability and accuracy of machine learning models. As mentioned before, if we implement the decision trees, sometimes the training set is overfitting. The reason why this happens is related to its processing mechanism. In the training process, every node will be split into two new nodes based on different conditions, so the model will be overfitting caused by fitting both parts. This is also the reason why the decision tree has a high variance. For the same decision tree, when new data comes in, the predicted results vary a lot. At this time, the Bootstrap method can help avoid overfitting.

The Bootstrap method is one of the resampling methods and is widely applicable. The resampling process can be shown as following procedures:

1. Choose a sample from population with a size of m as the original sample.
2. Generate a random sample from the data of original sample with replacement, meanwhile, let the newly generated sample has the same size m as the original sample has.
3. Repeat step 2 for b times.

For the random forest method, the emphasis is to figure out how many features should be trying to approach the best split. In the splitting process, we should avoid using the majority of all available predictors, because the Random Forest will be affected much by those strong predictors. If a strong predictor is utilized, the decision trees are correspondingly to be analogous to each other, which also shows a high-related relationship between predictors [6]. To avoid selecting the strong predictors, the Random Forest method stipulates that in every split, only the subsets of all predictors can be considered [1]. Assume that there are d predictors totally, the number of predictors in subsets with a denotation of N are suggested to satisfies the formula as follows.

In this project, the total number of predictors is denoted as d, which means only $\sqrt{d}$ or $d/3$ predictors are allowed to be considered in each node splitting process. So the number of considered predictors is defined as

$$N = \sqrt{d} \qquad or \qquad N = \frac{d}{3}$$

Generally, the process of building a random forest is listed as follows:

1. Generate a random sample using Bootstrap with a size of b.
2. Build a decision tree for this Bootstrap random sample.

3. Repeat the previous 2 steps for k times.

4. Collect results from all the decision trees and classify the data into a cluster with more votes among all aggregation results.

4.5 Feed-Forward Neural Network Model

Feed-Forward neural network (FNN) is one of models that commonly used in supervised learning case. The other well-known neural networks such as Conventional Neural Network (CNN) and Recurrent Neural Network (RNN), usually occurs as special cases in the FNN field. Since we are not analyzing image data in this project, we implement FNN model and take this model as a representative of all neural networks for research. Comparing to other Neural Network model, FNN is not limited to linear functions and it is able to give a reasonable approximation of linear functions, while linear model may be restricted to only linear functions [11].

Generally, the main goal of FNN is to achieve an approximation of certain functions [14]. For instance, there is a function that takes X as input and Y as output. FNN maps the input X to an outcome Y by constructing a neural network architecture. In a FNN model, there may be a bunch of neural layers but at least three layers. The simplest FNN model should contains an input layer, a hidden layer or called the second layer, and an output layer. In a more complex FNN model, only the number of hidden layers can be increased, the input layer and output layer can only be constructed once. Another special case of FNN is that the information can be stored within a neural network and there exist loops in the neural network architecture, which is called the Recurrent Neural Network (RNN) [11].

As what is named for FNN, the flow of information has a forward direction. To help fully understand the meaning of forward, we use the previous example that a function takes X as the input and Y as the output. X will be used to calculate and approach the next layer, then being calculated in turn until the FNN model gives an outcome Y.

From the definition of cost function (Definition 5), it is known that a cost function always shows the difference between the value of target outcomes and the predicted outcomes approximated by a chosen model. It can be said that the smaller the cost function, the more accurate the model. FNN's cost function aims to evaluate how well the neural network is as a whole instead of single neural.

The most commonly used cost function in FNN is to use the cross-entropy between the training data and the predictions as the cost function. The Cross-Entropy cost function is also called Binary Cross-Entropy or Bernoulli negative log-likelihood. Generally, in machine learning, Cross-Entropy cost function can be written as shown below:

$$C(W, B, S^r, E^r) = -\sum_j [E_j^r \ln(a_j^L) + (1 - E_j^r)\ln(1 - a_j^L)]$$

, where W represents the weight of neural network, B represents the bias of nueral network, S_r and E_r are the input and the designed output of a training samples.

In this project, we build a simple Feed-forward Neural Network model with only three layers. The first layer is the input layer, it takes d input and uses 'Relu' activation. The second layer uses the same 'Relu' activation as the input layer does. The third layer is the output layer, who uses the 'sigmoid' activation to predict a value in the range of [0,1].

Figure 8 below shows an example of 4 layer Feed-Forward Neural Network. This FNN has 1 input layer, 2 hidden layers, and 1 output layer. In each hidden layer, there are 6 nodes.

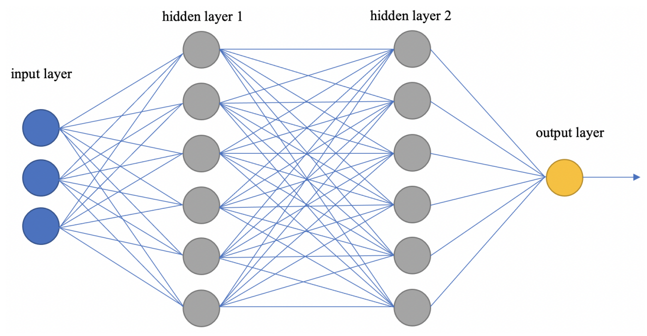


Figure 8: The architecture of FNN with 2 hidden layers.

4.6 XG boost Model

When it comes to the XG boost model, it's of great value to learn about its core algorithm, the Gradient Boosted Tree Algorithm. The Gradient Boosted Tree Algorithm is one of the supervised learning methods, which implements the approximation process by optimizing the loss function, thus optimizing the cost function. It's usually used when solving classification problems and regression problems.

What is different from other models is that the Gradient Boosted Algorithm can use multiple weak learners but provide a quite strong model eventually to solve the problem [8]. Generally, it is a sequential process of finding the best next model. It minimizes the overall prediction errors based on the previous model obtained, thus find the next model with fewer errors. The Gradient boosted algorithm will repeat this step until it can finally return a strong model with a global optimum prediction error. Every model involved in this optimizing process is called the weak learner. The weak learner used in Gradient Boosting is the Decision Tree [2].

Generally, the gradient boosted algorithm has following steps [7]:

1. Calculate the mean value of the target variable.

2. Use the formula below to calculate the residuals for every observation.

$$Residual = Actualvalue - Predictedvalue$$

3. To predict the residuals, we need to build a decision tree with several conditions. If there are more than one leaf node existed for an inside node, than we remove all leaf nodes of this inside node and assigned a new leaf node with the mean value of all previous leaf nodes. Thus making sure every inside node has only one leaf node.

4. Calculate the target price with the formula below:

$$Newpredictedvalue = Avg(Targetvalue) + learningrate \times residual$$

   , where residual is predicted by the decision tree in former steps.

5. Predict the label of the target variable based on the target value we got in step 4.

6. Calculate the new residuals.

7. Repeat step 3 to step 6, until the number of iterations reaches the number of estimators (hyperparameter) we set.

As to the XG boost model, it is a regularized form of Gradient boosting [8]. Comparing to Gradient boosting, the XG boost holds a higher performance. This is mainly reflected by the very little time it takes in the process of training data. The reason for its fast training pace is that the XG boost implements advanced regularization, $L1$ and $L2$.

Definition 6 (L1 and L2 regularization) *$L1$ and $L2$ regularization is two techniques that can be used to prevent model overfitting. Given that the loss with no regularization is denoted as below:*

$$Loss = Error(y, \hat{y})$$

Then the loss funciton with L1 and L2 regularization are shown in order respectively as below:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} \mid \omega_i \mid$$

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} \omega_i^2$$

5 Data

The data analyzed in this project is generated by several methods. We totally generate two kinds of dataset, which is independent dataset and dependent dataset. For each kind of data, there are two mechanisms to generate the predict values.

5.1 Generating independent data matrix X

As mentioned in the section: Method and Models, each dataset has a size of n and a dimension of d. The independent data means the dataset is contributed by n observations (X_i), which are vectors with the same size of $1 \times d$ and are generated randomly. This is to say, each X_i is independent of other vectors and every element in X_i is independent of other elements. To implement n d-dimentional-vector, a normal distribution is used because it describes sample distribution who is affected by a large number of random disturbances. Note that in this project, we set the mean to 0 and set the covariance to 1 to make a standard normal distribution. This approach is implemented with the Python package 'NumPy. Random', to fully make sure the randomness of data.

5.2 Generating dependent data matrix X

Definition 7 (Correlation) *Essentially, correlation is the measure of how two or more variables are related to one another.*

When it comes to generating dependent data [10], the relationship between elements in X_i is no longer independent, which means there is a correlation exists between two or more variables for each X_i. Instead of the Normal Distribution, a Normal Matrix Distribution is utilized to implement the dependencies of data. The Normal Matrix Distribution is detailed explained in the section: introduction and we will loosely follow here. The process of generating data with Normal Matrix Distribution needs three parameters, the mean denoted as M, the covariance among rows denoted as U, and the covariance among columns denoted as V. In order to control variables as much as possible when comparing independent data and dependent data, we keep setting the mean to 0. However, both form of mean and covariance are changed. In a dataset X that generated from Normal Matrix Distribution, the mean M is an $n \times d$ matrix full of 0, the M matrix is listed below:

$$M = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Covariance among rows U is a block matrix with size $n \times n$ has the form shown below:

$$U = \begin{bmatrix}
1 & \rho & \rho & \cdots & \rho & & 0 & 0 & 0 & \cdots & 0 \\
\rho & 1 & \rho & \cdots & \rho & & 0 & 0 & 0 & \cdots & 0 \\
\rho & \rho & 1 & \ddots & \vdots & & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \rho & & \vdots & \vdots & \vdots & \ddots & \vdots \\
\rho & \rho & \cdots & \rho & 1 & & 0 & 0 & 0 & \cdots & 0 \\
 & & & & & & & & & & \\
0 & 0 & 0 & \cdots & 0 & & 1 & \rho & \rho & \cdots & \rho \\
0 & 0 & 0 & \cdots & 0 & & \rho & 1 & \rho & \cdots & \rho \\
0 & 0 & 0 & \cdots & 0 & & \rho & \rho & 1 & \ddots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \vdots & & \vdots & \vdots & \ddots & \ddots & \rho \\
0 & 0 & 0 & \cdots & 0 & & 1 & \rho & \rho & \cdots & \rho
\end{bmatrix}$$

Covariance among columns V is an identity matrix with size $d \times d$, its form is listed below:

$$V = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

Then the covariance is obtained from the Kronecker product of U and V, with a size of $(n \times d) \times (n \times d)$ and is denoted as K. Noted that we have two values of rho in this approach, 0.1 and 0.9, for 0.1 represents a weak relationship and 0.9 represents a strong relationship between random variables. Additionally, since the U is formed as a block matrix, the value of n must be an even number so that it can be divided by 2 to represent half of the size in tems of a block matrix. This approach is implemented with the Python package 'NumPy. Random. Multivariate normal'.

5.3 Generating predict values for data matrix X

In this project, two mechanisms are utilized to generate predict values for both independent data and dependent data. To generate the predict value, we implement the probability of data by using the logit function and Bernoulli Distribution. In both mechanisms, we use the sigmoid function to calculate the probability. For a detailed explanation of the logit function and sigmoid function, we refer to the content explained in the Method part: Logistic regression, which we loosely follow here.

In the first mechanism, since the output of logistic sigmoid regression can only be approximated to 0 or 1, we implement an if-statement and set a threshold of 0.5 to classify the result into two groups. For the group with outputs that are larger than 0.5, we update the outputs as 1s, meanwhile, we update all the rest of outputs as 0s.

In the second mechanism, we implement Bernoulli distribution by using the preset Bernoulli method from Scipy package. This package takes the probability of data matrix X as input and generate the predict values as 1s or 0s automatically.

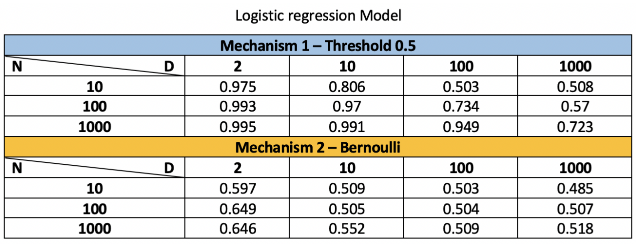
The reason why we retain both mechanisms is that during the process of testing models, the two mechanisms of generating predict values show a different result of accuracy.

6 Results

6.1 Accuracy result of independent data

For each of the 6 models, we provide the results from 2 mechanisms. As described in the section5.3: Generating predict values for data matrix X, the first mechanism uses a threshold of 0.5 to classify the results while the second mechanism utilizes the preset Bernoulli method to predict the results. In each mechanism, we also choose the values of n from $\{10, 100, 1000\}$ and d from $\{2, 10, 100, 1000\}$.

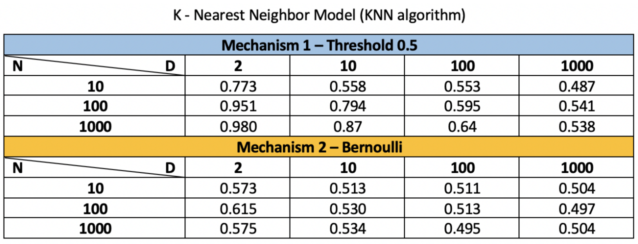
The following 6 figures show the prediction accuracy obtained by implementing the 6 models under 2 different mechanisms.



Logistic regression Model

Mechanism 1 – Threshold 0.5				
N D	2	10	100	1000
10	0.975	0.806	0.503	0.508
100	0.993	0.97	0.734	0.57
1000	0.995	0.991	0.949	0.723
Mechanism 2 – Bernoulli				
N D	2	10	100	1000
10	0.597	0.509	0.503	0.485
100	0.649	0.505	0.504	0.507
1000	0.646	0.552	0.509	0.518

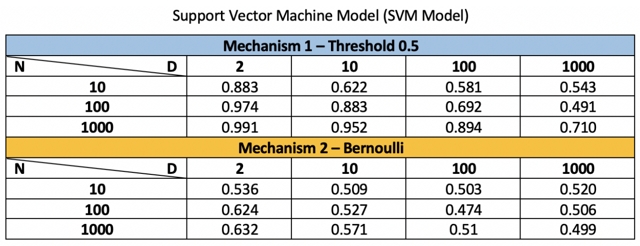
Figure 9: The prediction accuracy of Logistic Model under two mechanisms.



K - Nearest Neighbor Model (KNN algorithm)

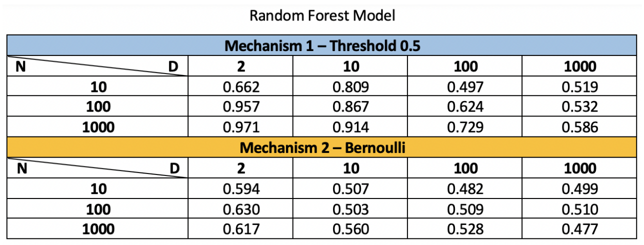
Mechanism 1 – Threshold 0.5				
N D	2	10	100	1000
10	0.773	0.558	0.553	0.487
100	0.951	0.794	0.595	0.541
1000	0.980	0.87	0.64	0.538
Mechanism 2 – Bernoulli				
N D	2	10	100	1000
10	0.573	0.513	0.511	0.504
100	0.615	0.530	0.513	0.497
1000	0.575	0.534	0.495	0.504

Figure 10: The prediction accuracy of KNN Model under two mechanisms.


Support Vector Machine Model (SVM Model)

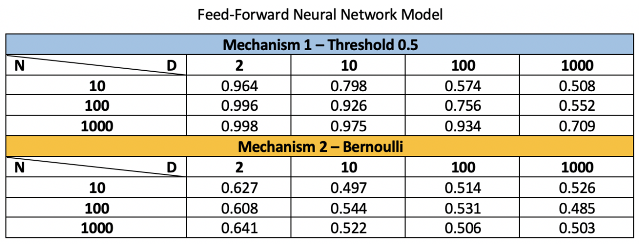
Mechanism 1 – Threshold 0.5				
N \ D	2	10	100	1000
10	0.883	0.622	0.581	0.543
100	0.974	0.883	0.692	0.491
1000	0.991	0.952	0.894	0.710
Mechanism 2 – Bernoulli				
N \ D	2	10	100	1000
10	0.536	0.509	0.503	0.520
100	0.624	0.527	0.474	0.506
1000	0.632	0.571	0.51	0.499

Figure 11: The prediction accuracy of SVM Model under two mechanisms.


Random Forest Model

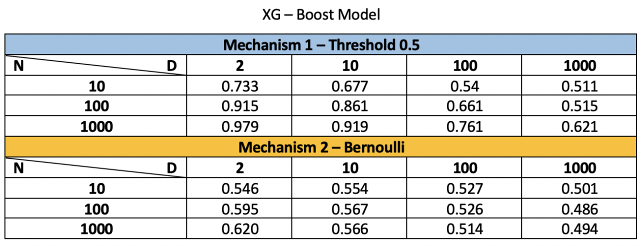
Mechanism 1 – Threshold 0.5				
N \ D	2	10	100	1000
10	0.662	0.809	0.497	0.519
100	0.957	0.867	0.624	0.532
1000	0.971	0.914	0.729	0.586
Mechanism 2 – Bernoulli				
N \ D	2	10	100	1000
10	0.594	0.507	0.482	0.499
100	0.630	0.503	0.509	0.510
1000	0.617	0.560	0.528	0.477

Figure 12: The prediction accuracy of Random Forest Model under two mechanisms.


Feed-Forward Neural Network Model

Mechanism 1 – Threshold 0.5				
N \ D	2	10	100	1000
10	0.964	0.798	0.574	0.508
100	0.996	0.926	0.756	0.552
1000	0.998	0.975	0.934	0.709
Mechanism 2 – Bernoulli				
N \ D	2	10	100	1000
10	0.627	0.497	0.514	0.526
100	0.608	0.544	0.531	0.485
1000	0.641	0.522	0.506	0.503

Figure 13: The prediction accuracy of Feed-Forward Nerual Network Model under two mechanisms.


XG – Boost Model

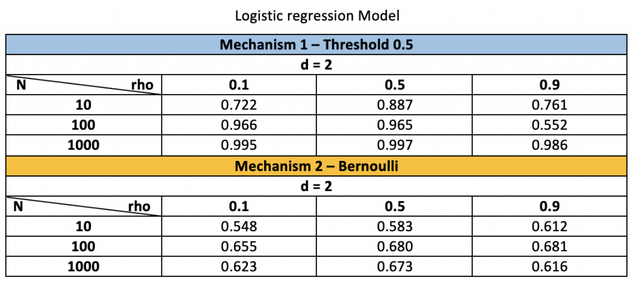
Mechanism 1 – Threshold 0.5				
N \ D	2	10	100	1000
10	0.733	0.677	0.54	0.511
100	0.915	0.861	0.661	0.515
1000	0.979	0.919	0.761	0.621
Mechanism 2 – Bernoulli				
N \ D	2	10	100	1000
10	0.546	0.554	0.527	0.501
100	0.595	0.567	0.526	0.486
1000	0.620	0.566	0.514	0.494

Figure 14: The prediction accuracy of XG Boost Model under two mechanisms.

6.2 Accuracy result of dependent data

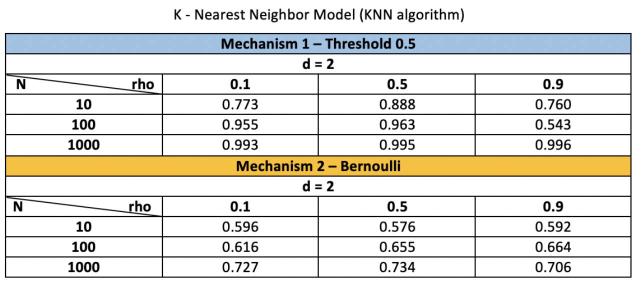
As for the prediction result for dependent data, we still provide the results from 2 mechanisms. In actual operation, we are only able to collect results with a small dimension due to the limit of computing performance of the device. So, for each of the 6 models, we keep setting the dimension d to a small value as 2 and choose the values of n from $\{10, 100, 1000\}$. In addition, we introduced a new parameter ρ when generating the data, so the parameter ρ should be also taken into consideration when comparing the prediction results. The value of ρ is chosen from $\{0.1, 0.9\}$, where 0.1 represents a weak correlation, 0.9 represents a strong correlation instead.

The following figures illustrate the prediction accuracy obtained by implementing the 6 models under 2 different mechanisms, also with different ρ.


Logistic regression Model

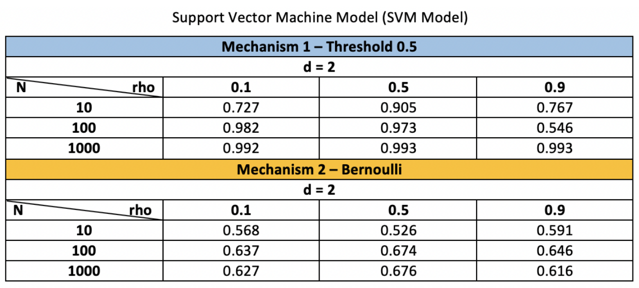
Mechanism 1 – Threshold 0.5			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.722	0.887	0.761
100	0.966	0.965	0.552
1000	0.995	0.997	0.986
Mechanism 2 – Bernoulli			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.548	0.583	0.612
100	0.655	0.680	0.681
1000	0.623	0.673	0.616

Figure 15: The prediction accuracy of Logistic Model under two mechanisms with $d = 2$.


K - Nearest Neighbor Model (KNN algorithm)

Mechanism 1 – Threshold 0.5			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.773	0.888	0.760
100	0.955	0.963	0.543
1000	0.993	0.995	0.996
Mechanism 2 – Bernoulli			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.596	0.576	0.592
100	0.616	0.655	0.664
1000	0.727	0.734	0.706

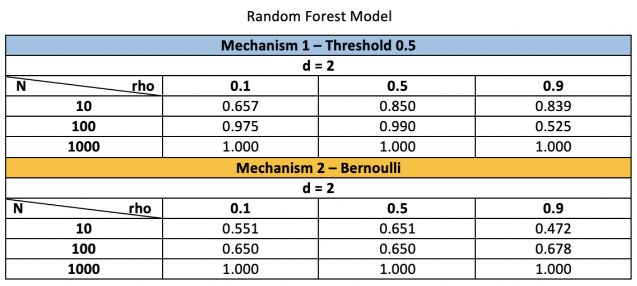
Figure 16: The prediction accuracy of KNN Model under two mechanisms with $d = 2$.



Support Vector Machine Model (SVM Model)

Mechanism 1 – Threshold 0.5			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.727	0.905	0.767
100	0.982	0.973	0.546
1000	0.992	0.993	0.993
Mechanism 2 – Bernoulli			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.568	0.526	0.591
100	0.637	0.674	0.646
1000	0.627	0.676	0.616

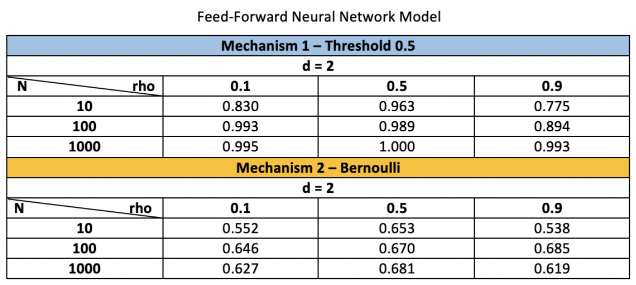
Figure 17: The prediction accuracy of SVM Model under two mechanisms with $d = 2$.



Random Forest Model

Mechanism 1 – Threshold 0.5			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.657	0.850	0.839
100	0.975	0.990	0.525
1000	1.000	1.000	1.000
Mechanism 2 – Bernoulli			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.551	0.651	0.472
100	0.650	0.650	0.678
1000	1.000	1.000	1.000

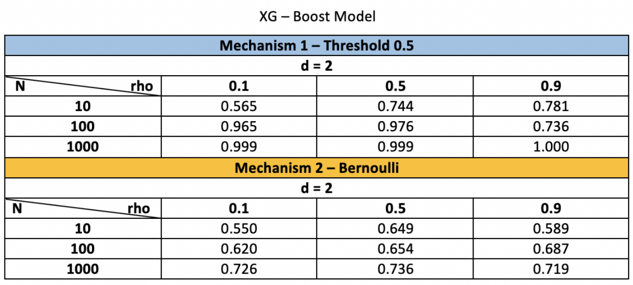
Figure 18: The prediction accuracy of Random Forest Model under two mechanisms with $d = 2$.



Feed-Forward Neural Network Model

Mechanism 1 – Threshold 0.5			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.830	0.963	0.775
100	0.993	0.989	0.894
1000	0.995	1.000	0.993
Mechanism 2 – Bernoulli			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.552	0.653	0.538
100	0.646	0.670	0.685
1000	0.627	0.681	0.619

Figure 19: The prediction accuracy of Feed-Forward Nerual Network Model under two mechanisms with $d = 2$.

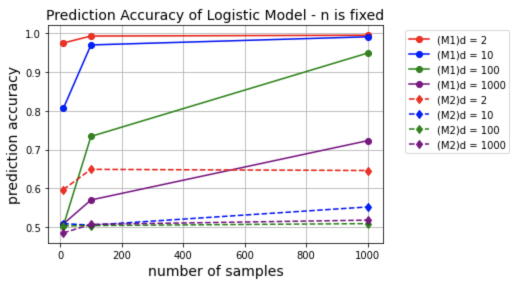

XG – Boost Model

Mechanism 1 – Threshold 0.5			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.565	0.744	0.781
100	0.965	0.976	0.736
1000	0.999	0.999	1.000
Mechanism 2 – Bernoulli			
d = 2			
N \ rho	0.1	0.5	0.9
10	0.550	0.649	0.589
100	0.620	0.654	0.687
1000	0.726	0.736	0.719

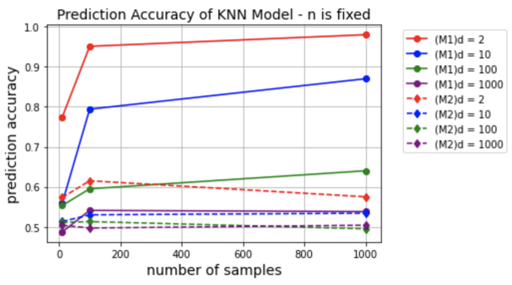
Figure 20: The prediction accuracy of XG Boost Model under two mechanisms with $d = 2$.

7 Discussion

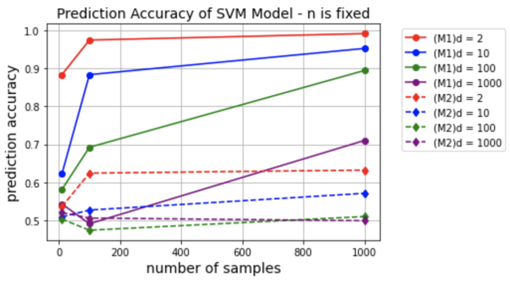
To fully analyze the influence of each parameters brought to the models' prediction accuracy, the discussion about accuracy results can be divided into 6 groups. The detailed group of discussion is listed below as well 3 overall figures of accuracy results.



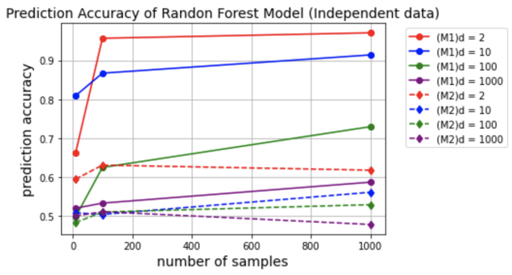
(a) Logistic Model with independent data



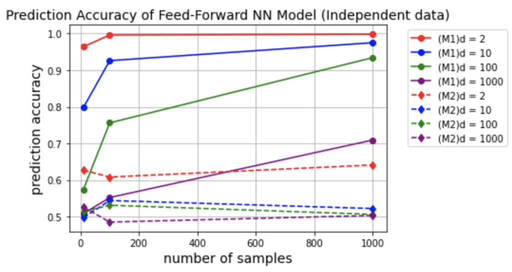
(b) KNN Model with independent data



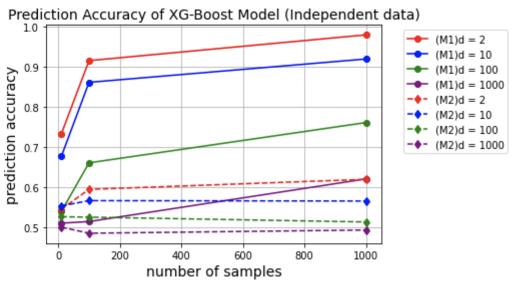
(c) SVM Model with independent data



(d) Random Forest Model with independent data

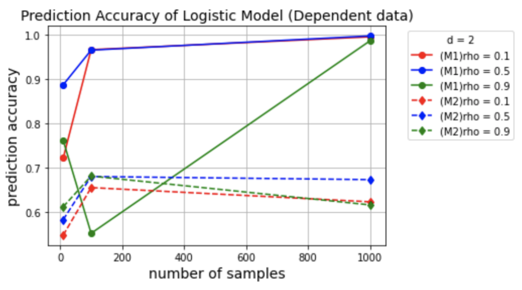


(e) Feed-Forward Neural Network Model with independent data

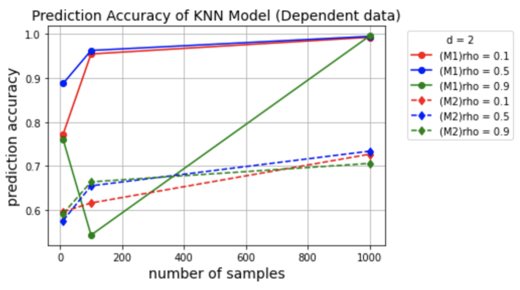


(f) XG-Boost Model with independent data

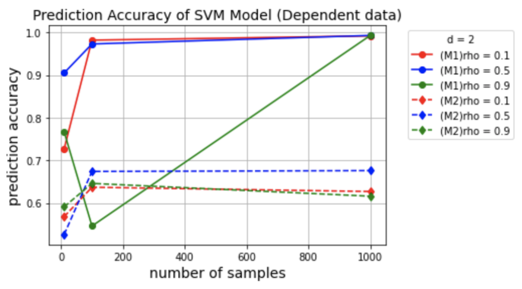
Figure 21: Line charts of accuracy results of 6 models by using the independent data.



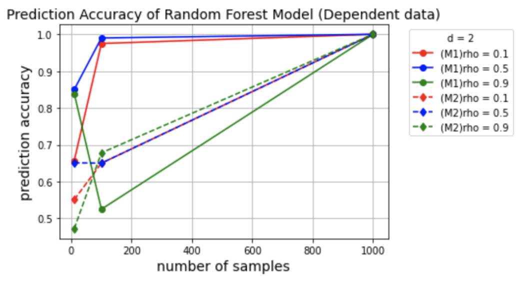
(a) Logistic Model with Dependent data



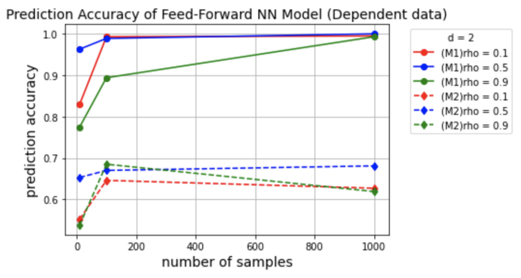
(b) KNN Model with Dependent data



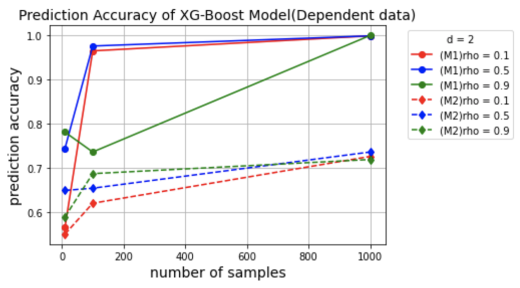
(c) SVM Model with Dependent data



(d) Random Forest Model with Dependent data



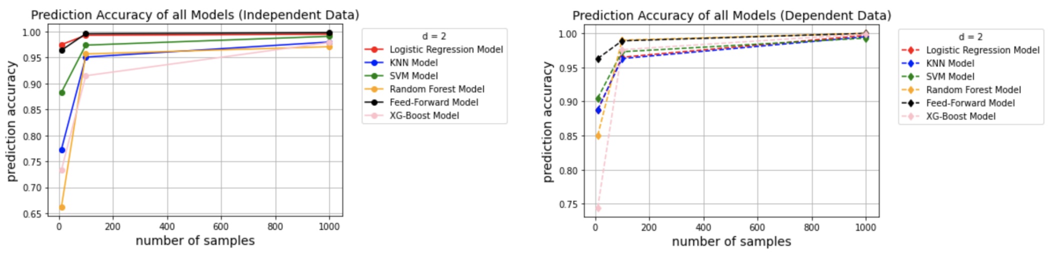
(e) Feed-Forward Neural Network Model with Dependent data



(f) XG-Boost Model with Dependent data

Figure 22: Line charts of accuracy results of 6 models by using the dependent data.

Figure 21 shows the trend of prediction accuracy changes brought about by inputting different parameter n and parameter d when using 6 models with independent data. The solid line represents the trend by using mechanism 1 (threshold) to compute predict values, while the dash lines represent the trend by using mechanism 2 (Bernoulli method). What's more, the red, blue, green, and purple lines represent different dimensions respectively. Similarly, figure 22 shows that when using dependent data, the trend of prediction accuracy changes brought by inputting different parameter n and parameter ρ in 6 models. The legend with this figure is similar to figure 21 except that the different colors represent different correlation values ρ.



(a) Prediction accuracy of all models with Independent data

(b) Prediction accuracy all models with Dependent data

Figure 23: Put all accuracy with the same data type together.

Figure 23 above shows the overall comparison of the prediction accuracy of models. The two charts are separated by gathering the accuracy obtained from the same data type. With regard to these two charts, we set a fixed d that $d = 2$, and choose the mean correlation coefficient ρ that $\rho = 0.5$, while $n \in \{10, 100, 1000\}$.

7.1 Group 1

Given that we use independent data with fixed n and increasing d, and both mechanisms of predicting value are considered.

All 6 accuracy charts in figure 21 show a similar trend that the accuracy keeps decreasing as the number of dimensions gets larger. In figure 21 – chart (c), when the number of samples n is less than 100, the accuracy is even below 0.8, which is smaller than the accuracy with a larger dimension such as $d = 10$. So, we can't say the smallest dimension will result in optimal prediction accuracy, but a decreasing dimension will result in more loss of prediction accuracy. In addition, the accuracy under two mechanisms keeps the same trend while n is fixed and d increases. Generally, the accuracy under mechanism 1 is larger than the mechanism 2 overall.

7.2 Group 2

Given that we use independent data with increasing n and fixed d, and both mechanisms of predicting value are considered.

Under mechanism 1, all 6 accuracy charts in figure 21 show a similar and obvious trend that the prediction accuracy becomes lager as the number of samples gets larger. However, when under mechanism 2, the accuracy trend of 6 models doesn't have a unified way of changing, in figure 21 – chart (a),(c), and (e), the peak point is reached when $n = 100$ instead of $n = 1000$, while in the rest of chart (b),(d), and (f), the increasing trend of accuracy is easily observed. So, under mechanism 1, the increase of number of samples n and the fixed number of dimensions will result in a corresponding increase of prediction accuracy. But under mechanism 2, the increase of n doesn't always lead to the increase of prediction accuracy.

7.3 Group 3

Given that we use independent data with $d = 2$ and mechanism 1.

Because in discussion group 1 and group 2, we found that mechanism 1 always provides a better prediction accuracy, so we use mechanism 1 in this overall comparison. Figure 23 – chart (a) shows the performance of 6 models when they predict with independent data. Apparently, the Logistic Regression Model has the best accuracy when $10 \leq n \leq 100$, and the Feed-forward Neural Network Model has the best accuracy when $100 \leq n \leq 1000$. Among all the models, the Random Forest Model has the sharpest increase when n increases from 10 to 100. When $100 \leq n \leq 1000$, the accuracy of all models has a flat increase trend. Overall, all prediction accuracy is improved as the number of samples n increases.

7.4 Group 4

Given that we use dependent data with fixed n and increasing ρ, and both mechanisms of predicting value are considered.

Under mechanism 1, by observing figure 22, the increase of ρ always brought the loss of accuracy when $n \leq 10$ and $100 \leq n$. However, $n = 100$ is a special case since the accuracy of other models will reach a valley at this time except for the accuracy of the Feed-Forward Neural Network Model. Meanwhile, $\rho = 0.5$ always reflects the highest prediction accuracy in all charts of figure 22. Under mechanism 2, the accuracy trend doesn't keep decreasing when ρ increases. On the contrary, the $\rho = 0.5$ always reflects the highest accuracy. It's hard to say that with a stronger relationship between random variables, the accuracy is more likely to be higher. So, under mechanism 1, when the ρ is small like 0.1 and is large like 0.9, the accuracy will increase as the ρ increases, meanwhile, when $\rho = 0.5$, the accuracy always reaches a peak of accuracy. Under mechanism 2, when n is around 100, a larger correlation coefficient ρ results in higher accuracy. In other cases, the chart can't reflect an obvious conclusion.

7.5 Group 5

Given that we use dependent data with increasing n and fixed ρ, and both mechanisms of predicting value are considered.

What is similar to the conclusions made in group 5 is that $n = 100$ is still a special case. Under mechanism 1, when $\rho = 0.1$ or $\rho = 0.5$, the accuracy of all models has an obvious increasing trend, which can be observed in figure 22. When $\rho = 0.9$, the prediction accuracy will reach a valley except for the accuracy chart of the Feed-Forward Neural Network Model. Under mechanism 2, in figure 22 – chart (b), (d), and (e), the accuracy of models shows an apparent increasing trend with an increasing n. As for figure 22 – chart (a), (c), and (e), when $10 \leq n \leq 100$, the accuracy increases sharply, however, when $100 \leq n$, the accuracy has a slightly decreasing trend.

7.6 Group 6

Given that we use dependent data with $d = 2$, $\rho = 0.5$, and mechanism 1.

The reason for choosing mechanism 1 is the same as the reason explained in discussion group 3 and we will loosely mention it here. Figure 23 – chart (b) shows the performance of the 6 models when they predict with the dependent data. Apparently, the Feed-forward Neural Network Model has the best accuracy with all values of $n \in \{10, 100, 1000\}$. Among all the models, the XG-Boost Model has the sharpest increase when n increases from 10 to 100. When $100 \leq n \leq 1000$, the accuracy of all models has a flat increase trend. Overall, all prediction accuracy is improved as the number of samples n increases.

8 Conclusion

In this thesis, we introduced the background knowledge of Multivariate Normal Distribution and Matrix Normal Distribution. Next, we introduce 6 commonly used machine learning models utilized to provide the predicted value. Additionally, the processes of generating independent and dependent datasets are detailed explained. Then we collect accuracy results from 6 models by input different parameters for both datasets. Ultimately, we discuss the conclusions made by observing the charts of accuracy trend into 6 different groups through the method of controlling variates, such as dataset types, number of samples, number of dimensions, and correlation coefficient.

9 Reference List

References

[1] Biau, G. Scornet, E. 2016. A random forest guided tour. Test252197–227.

[2] Chen, T. Guestrin, C. 2016. Xgboost: A scalable tree boosting system. Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (785–794).

[3] Kleinbaum, DG. Klein, M. 2010. Introduction to logistic regression Introduction to logistic regression. Logistic regression Logistic regression (1–39). Springer.

[4] A logical calculus of the ideas immanent in nervous activity. 1943. The bulletin of mathematical biophysics. https://doi.org/10.1007/BF02478259

[5] McDonald, C. 2017. Machine learning fundamentals (i): Cost functions and gradient descent. Towards Data Science27.

[6] Murthy, KVS. Salzberg, SL. 1995. On growing better decision trees from data . Citeseer.

[7] Natekin, A. Knoll, A. 2013. Gradient boosting machines, a tutorial. Frontiers in neurorobotics721.

[8] Nielsen, D. 2016. Tree boosting with xgboost. NTNU Norwegian University of Science and Technology.

[9] Park, H. 2013. An introduction to logistic regression: from basic concepts to interpretation with particular attention to nursing domain. Journal of Korean Academy of Nursing432154–164.

[10] Racine, J. 2000. Consistent cross-validatory model-selection for dependent data: hv-block cross-validation. Journal of econometrics99139–61.

[11] Sazlı, MH. 2006. A brief review of feed-forward neural networks.

[12] Stecanella, B. 2017. An Introduction to Support Vector Machines (SVM). Monkey Learn, https://monkeylearn. com/blog/introduction-tosupport-vector-machines-svm.

[13] Subramanian, D. 2019. A simple introduction to K-Nearest Neighbors Algorithm. Towards Data Science.

[14] Svozil, D., Kvasnicka, V. Pospichal, J. 1997. Introduction to multi-layer feed-forward neural networks. Chemometrics and intelligent laboratory systems39143–62.

[15] Tong, YL. 2012. The multivariate normal distribution. Springer Science & Business Media.

10 Appendix

Appendix 1: Independent Variable

```
# -*- coding: utf-8 -*-
"""Independent_variable.ipynb

Automatically_generated_by_Colaboratory.
"""


import numpy as np
import numpy.random as nr
import random
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import math
from scipy import stats


#method to generate Data matrix X with n Xi, each Xi is a d-dimentional vector
def GenerateData(n, d, v): #n:vector number of X_n; d: number of dimensions of X
  nr.seed()
  X = np.array(nr.normal(0,1,v))
  for i in range (0, n-1):
    newVec = np.array(nr.normal(0,1,v)) #mean = 0, cov = 1, size = 1*d
    X = np.vstack((X,newVec))
  return X  #X is a numpy.ndarray
```

```
def GenerateTarget(n, X_n, beta):
  Y = []
  p = []
  np.random.seed(123)
  for i in range (0, n):
    p_i = 1/(1+math.exp(-np.dot(X_n[i],beta[0])))
    p.append(p_i)
  for j in range (0, n):
    if p[j] >= 0.5:
      Y.append(1)
    else:
      Y.append(0)
  return Y

"""**Step 1 - 2**: Creating the method to generate Y_n using **Bernoulli Distrib
def GenerateTarget_B(n, X_n, beta):
  Y = []
  np.random.seed(123)
  for i in range (0, n):
    p_i = 1/(1+math.exp(-np.dot(X_n[i],beta[0])))
    bernoulliDist = stats.bernoulli(p_i)
    #Y.append(bernoulliDist.rvs(1))
    Y.append(bernoulliDist.rvs(1))
    np.ravel(Y)
  return Y
```

```
"""#␣**1.␣Logistic␣Regression**"""

def GetLogisticResult(n, d, v):

  X_n = GenerateData(n, d, v)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget(n,X_n,beta)

  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set

  test_Y = GenerateTarget(1000,test_X, beta)

  logisticRegressionModel = LogisticRegression()

  logisticRegressionModel.fit(X_n,Y_n)

  pred_y = logisticRegressionModel.predict(test_X)

  print('Accuracy:␣',metrics.accuracy_score(test_Y, pred_y))


def GetLogisticResult_B(n, d, v):

  X_n = GenerateData(n, d, v)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget_B(n,X_n,beta)

  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set

  test_Y = GenerateTarget_B(1000,test_X, beta)

  logisticRegressionModel = LogisticRegression()

  logisticRegressionModel.fit(X_n,Y_n)

  pred_y = logisticRegressionModel.predict(test_X)

  print('Accuracy:␣',metrics.accuracy_score(test_Y, pred_y))


"""Test␣X_n␣with␣different␣n␣and␣d␣by␣using␣logistic␣regression␣model."""
```

```
GetLogisticResult(10,2, (1,2))
GetLogisticResult(100,2, (1,2))
GetLogisticResult(1000,2, (1,2))

GetLogisticResult_B(10,2, (1,2))
GetLogisticResult_B(100,2,(1,2))
GetLogisticResult_B(1000,2, (1,2))

GetLogisticResult(10,10, (1,10))
GetLogisticResult(10,100,(1,100))
GetLogisticResult(10,1000, (1,1000))

GetLogisticResult(100,10,(1,10))
GetLogisticResult(100,100,(1,100))
GetLogisticResult(100,1000,(1,1000))

GetLogisticResult(1000,10,(1,10))
GetLogisticResult(1000,100,(1,100))
GetLogisticResult(1000,1000,(1,1000))

GetLogisticResult_B(10,10, (1,10))
GetLogisticResult_B(10,100,(1,100))
GetLogisticResult_B(10,1000, (1,1000))

GetLogisticResult_B(100,10,(1,10))
GetLogisticResult_B(100,100,(1,100))
GetLogisticResult_B(100,1000,(1,1000))


GetLogisticResult_B(1000,10,(1,10))
GetLogisticResult_B(1000,100,(1,100))
GetLogisticResult_B(1000,1000,(1,1000))


"""## **2. Nearest Neibour Model(KNN algorithm)**"""
from sklearn.neighbors import KNeighborsClassifier
def GetKNNResult(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget(n,X_n,beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget(1000,test_X, beta)
  KNNclassifier = KNeighborsClassifier(n_neighbors=5)
  KNNclassifier.fit(X_n, Y_n)
  pred_y = KNNclassifier.predict(test_X)
  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))


def GetKNNResult_B(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget_B(n,X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
```

```
    test_Y = GenerateTarget_B(1000,test_X, beta)
    KNNclassifier = KNeighborsClassifier(n_neighbors=5)
    KNNclassifier.fit(X_n, Y_n)
    pred_y = KNNclassifier.predict(test_X)
    print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

GetKNNResult(10,2, (1,2))
GetKNNResult(100,2, (1,2))
GetKNNResult(1000,2, (1,2))

GetKNNResult_B(10,2, (1,2))
GetKNNResult_B(100,2,(1,2))
GetKNNResult_B(1000,2, (1,2))

GetKNNResult(10,10, (1,10))
GetKNNResult(10,100,(1,100))
GetKNNResult(10,1000, (1,1000))

GetKNNResult(100,10,(1,10))
GetKNNResult(100,100,(1,100))
GetKNNResult(100,1000,(1,1000))

GetKNNResult(1000,10,(1,10))
GetKNNResult(1000,100,(1,100))
GetKNNResult(1000,1000,(1,1000))
```

```
GetKNNResult_B(10,10, (1,10))
GetKNNResult_B(10,100,(1,100))
GetKNNResult_B(10,1000, (1,1000))


GetKNNResult_B(100,10,(1,10))
GetKNNResult_B(100,100,(1,100))
GetKNNResult_B(100,1000,(1,1000))


GetKNNResult_B(1000,10,(1,10))
GetKNNResult_B(1000,100,(1,100))
GetKNNResult_B(1000,1000,(1,1000))


"""## **3. Support Vector Machine (SVM) Model**"""
from sklearn.svm import SVC
def GetSVMResult(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget(1000, test_X, beta)
  SVMclassifier = SVC(kernel='rbf', random_state = 1)
  SVMclassifier.fit(X_n,Y_n)
  pred_y = SVMclassifier.predict(test_X)
  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))
```

```
def GetSVMResult_B(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget_B(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget_B(1000, test_X, beta)
  SVMclassifier = SVC(kernel='rbf', random_state = 1)
  SVMclassifier.fit(X_n,Y_n)
  pred_y = SVMclassifier.predict(test_X)
  print('Accuracy:␣',metrics.accuracy_score(test_Y, pred_y))

GetSVMResult(10,2, (1,2))
GetSVMResult(100,2, (1,2))
GetSVMResult(1000,2, (1,2))

GetSVMResult_B(10,2, (1,2))
GetSVMResult_B(100,2,(1,2))
GetSVMResult_B(1000,2, (1,2))

GetSVMResult(10,10, (1,10))
GetSVMResult(10,100,(1,100))
GetSVMResult(10,1000, (1,1000))

GetSVMResult(100,10,(1,10))
GetSVMResult(100,100,(1,100))

GetSVMResult(100,1000,(1,1000))


GetSVMResult(1000,10,(1,10))

GetSVMResult(1000,100,(1,100))

GetSVMResult(1000,1000,(1,1000))


GetSVMResult_B(10,10, (1,10))

GetSVMResult_B(10,100,(1,100))

GetSVMResult_B(10,1000, (1,1000))


GetSVMResult_B(100,10,(1,10))

GetSVMResult_B(100,100,(1,100))

GetSVMResult_B(100,1000,(1,1000))


GetSVMResult_B(1000,10,(1,10))

GetSVMResult_B(1000,100,(1,100))

GetSVMResult_B(1000,1000,(1,1000))


"""## **4. Random Forest Model**"""

from sklearn.ensemble import RandomForestClassifier

def GetRandomForestResult(n, d, v):

  X_n = GenerateData(n, d, v)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget(1000, test_X, beta)
  RFclassifier = RandomForestClassifier(n_estimators=100) #set the number of tre
  RFclassifier.fit(X_n,Y_n)
  pred_y = RFclassifier.predict(test_X)
  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

def GetRandomForestResult_B(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget_B(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget_B(1000, test_X, beta)
  RFclassifier = RandomForestClassifier(n_estimators=100) #set the number of tre
  RFclassifier.fit(X_n,Y_n)
  pred_y = RFclassifier.predict(test_X)
  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

GetRandomForestResult(10,2, (1,2))
GetRandomForestResult(100,2, (1,2))
GetRandomForestResult(1000,2, (1,2))

GetRandomForestResult_B(10,2, (1,2))
GetRandomForestResult_B(100,2,(1,2))
GetRandomForestResult_B(1000,2, (1,2))
```

```
GetRandomForestResult(10,10,  (1,10))
GetRandomForestResult(10,100,(1,100))
GetRandomForestResult(10,1000,  (1,1000))

GetRandomForestResult(100,10,(1,10))
GetRandomForestResult(100,100,(1,100))
GetRandomForestResult(100,1000,(1,1000))

GetRandomForestResult(1000,10,(1,10))
GetRandomForestResult(1000,100,(1,100))
GetRandomForestResult(1000,1000,(1,1000))

GetRandomForestResult_B(10,10,  (1,10))
GetRandomForestResult_B(10,100,(1,100))
GetRandomForestResult_B(10,1000,  (1,1000))

GetRandomForestResult_B(100,10,(1,10))
GetRandomForestResult_B(100,100,(1,100))
GetRandomForestResult_B(100,1000,(1,1000))

GetRandomForestResult_B(1000,10,(1,10))
GetRandomForestResult_B(1000,100,(1,100))
GetRandomForestResult_B(1000,1000,(1,1000))
```

```
"""## **5. Feed−Forward Neural Network**"""
import keras
from keras.models import Sequential,Input,Model
from keras.layers import Conv2D
from keras.layers import Dense, Flatten, Activation

def GetFFNNResult(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget(1000, test_X, beta)
  X_n = np.array(X_n)
  Y_n = np.array(Y_n)
  test_X = np.array(test_X)
  test_Y = np.array(test_Y)
  #Build the feedforward NN model
  NNmodel = Sequential()
  NNmodel.add(Dense(64, input_shape = v, activation = "relu"))  #The first layer
  NNmodel.add(Dense(32, activation='relu'))    #The second layer
  NNmodel.add(Dense(1, activation='sigmoid'))      #Output layer
  #Compile the feed−forward NN model
  NNmodel.compile(loss = "binary_crossentropy", optimizer = "Adam",metrics=['acc
  NNmodel.fit(X_n, Y_n, epochs=100, batch_size=10, verbose=0)
  _, accuracy = NNmodel.evaluate(test_X, test_Y)
```

```
    print('Accuracy: %.2f' % (accuracy))


def GetFFNNResult_B(n, d, v):
    X_n = GenerateData(n, d, v)
    beta = np.random.dirichlet(np.ones(d),size=1)
    Y_n = GenerateTarget_B(n, X_n, beta)
    test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
    test_Y = GenerateTarget_B(1000, test_X, beta)
    X_n = np.array(X_n)
    Y_n = np.array(Y_n)
    test_X = np.array(test_X)
    test_Y = np.array(test_Y)
    #Build the feedforward NN model
    NNmodel = Sequential()
    NNmodel.add(Dense(64, input_shape = v, activation = "relu"))  #The first layer
    NNmodel.add(Dense(32, activation='relu'))    #The second layer
    NNmodel.add(Dense(1, activation='sigmoid'))      #Output layer
    #Compile the NN model
    NNmodel.compile(loss = "binary_crossentropy", optimizer = "Adam",metrics=['acc
    NNmodel.fit(X_n, Y_n, epochs=100, batch_size=10, verbose=0)
    _, accuracy = NNmodel.evaluate(test_X, test_Y)
    print('Accuracy: %.2f' % (accuracy))


GetFFNNResult(10,2, (1,2))
GetFFNNResult(100,2, (1,2))
GetFFNNResult(1000,2, (1,2))


GetFFNNResult_B(10,2, (1,2))

GetFFNNResult_B(100,2,(1,2))

GetFFNNResult_B(1000,2, (1,2))


GetFFNNResult(10,10, (1,10))

GetFFNNResult(10,100,(1,100))

GetFFNNResult(10,1000, (1,1000))


GetFFNNResult(100,10,(1,10))

GetFFNNResult(100,100,(1,100))

GetFFNNResult(100,1000,(1,1000))


GetFFNNResult(1000,10,(1,10))

GetFFNNResult(1000,100,(1,100))

GetFFNNResult(1000,1000,(1,1000))


GetFFNNResult_B(10,10, (1,10))

GetFFNNResult_B(10,100,(1,100))

GetFFNNResult_B(10,1000, (1,1000))


GetFFNNResult_B(100,10,(1,10))

GetFFNNResult_B(100,100,(1,100))

GetFFNNResult_B(100,1000,(1,1000))
GetFFNNResult_B(1000,10,(1,10))
GetFFNNResult_B(1000,100,(1,100))
GetFFNNResult_B(1000,1000,(1,1000))

"""## **6. XGboost Model**"""
from xgboost import XGBClassifier
def GetXGboostResult(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget(1000, test_X, beta)
  XGboost_classifier = XGBClassifier()
  XGboost_classifier.fit(np.asarray(X_n),np.asarray(Y_n))
  pred_y = XGboost_classifier.predict(np.asarray(test_X))
  predictions = [round(value) for value in pred_y]
  print('Accuracy: ',metrics.accuracy_score(test_Y, predictions))

def GetXGboostResult_B(n, d, v):
  X_n = GenerateData(n, d, v)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget_B(n, X_n, beta)
  test_X = GenerateData(1000,d,v) #to get accurate results in accuracy test, set
  test_Y = GenerateTarget_B(1000, test_X, beta)
  XGboost_classifier = XGBClassifier()
  XGboost_classifier.fit(np.asarray(X_n),np.asarray(Y_n))
  pred_y = XGboost_classifier.predict(np.asarray(test_X))
  predictions = [round(value) for value in pred_y]
  print('Accuracy: ',metrics.accuracy_score(test_Y, predictions))

GetXGboostResult(10,2, (1,2))
GetXGboostResult(100,2, (1,2))
GetXGboostResult(1000,2, (1,2))

GetXGboostResult_B(10,2, (1,2))
GetXGboostResult_B(100,2,(1,2))
GetXGboostResult_B(1000,2, (1,2))

GetXGboostResult(10,10, (1,10))
GetXGboostResult(10,100,(1,100))
GetXGboostResult(10,1000, (1,1000))

GetXGboostResult(100,10,(1,10))
GetXGboostResult(100,100,(1,100))
GetXGboostResult(100,1000,(1,1000))

GetXGboostResult(1000,10,(1,10))
GetXGboostResult(1000,100,(1,100))
GetXGboostResult(1000,1000,(1,1000))
```

```
GetXGboostResult_B(10,10,  (1,10))
GetXGboostResult_B(10,100,(1,100))
GetXGboostResult_B(10,1000,  (1,1000))

GetXGboostResult_B(100,10,(1,10))
GetXGboostResult_B(100,100,(1,100))
GetXGboostResult_B(100,1000,(1,1000))

GetXGboostResult_B(1000,10,(1,10))
GetXGboostResult_B(1000,100,(1,100))
GetXGboostResult_B(1000,1000,(1,1000))
```

Appendix 2: Dependent Variable

```
# -*- coding: utf-8 -*-
"""Dependent variables (Thesis Appendix).ipynb

Automatically generated by Colaboratory.
"""

import numpy as np
import numpy.random as nr
import random
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import math
from scipy import stats


def GenerateData_D(n, d, rho):
  #n: number of X_n matrix in X
  #d: number of dimensions of X_n matrix (columus of X_n vector)
  #rho: correlation coefficient
  nr.seed(123)
  M = np.full(n*d, 0) #mean vector, size n*d
  U11 = np.full((n // 2, n // 2), rho)
  U22 = np.full((n // 2, n // 2), rho)
  U12 = np.zeros((n // 2, n // 2))
  U21 = np.zeros((n // 2, n // 2))
  for i in range(n // 2):
    U11[i, i] = 1
    U22[i, i] = 1
  U = np.block([[U11, U12], [U21, U22]])#determing covariance among rows
  V = np.identity(d) #determing covariance among columns
  #Calculate Kronecker product
  K = np.kron(U,V) #size: (n*d) * (n*d)
  vector = np.random.multivariate_normal(M,K,(1,1))
  X = vector.reshape(n,d)
  return X  #X is a numpy.ndarray


def GenerateTarget(n, X_n, beta):
  Y = []
  p = []
  np.random.seed(123)
  for i in range (0, n):
    p_i = 1/(1+math.exp(-np.dot(X_n[i],beta[0])))
    p.append(p_i)
  for j in range (0, n):
    if p[j] >= 0.5:
      Y.append(1)
    else:
      Y.append(0)
  return Y


def GenerateTarget_B(n, X_n, beta):
  Y = []
  np.random.seed(123)
  for i in range (0, n):
    p_i = 1/(1+math.exp(-np.dot(X_n[i],beta[0])))
    bernoulliDist = stats.bernoulli(p_i)
    Y.append(bernoulliDist.rvs(1))
    np.ravel(Y)
  return Y

def GetLogisticResult(n, d, rho):
  X_n = GenerateData_D(n, d, rho)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget(n,X_n,beta)
  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test
  test_Y = GenerateTarget(1000,test_X, beta)
  logisticRegressionModel = LogisticRegression()
  logisticRegressionModel.fit(X_n,Y_n)
  pred_y = logisticRegressionModel.predict(test_X)
  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

def GetLogisticResult_B(n, d, rho):
  X_n = GenerateData_D(n, d, rho)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget_B(n,X_n,beta)
  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test
  test_Y = GenerateTarget_B(1000,test_X, beta)
```

```
    logisticRegressionModel = LogisticRegression()
    logisticRegressionModel.fit(X_n,Y_n)
    pred_y = logisticRegressionModel.predict(test_X)
    print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

"""Test X_n with different n and d by using logistic regression model."""
#mechanism 1
GetLogisticResult(10,2, 0.1)
GetLogisticResult(100,2, 0.1)
GetLogisticResult(1000,2, 0.1)

GetLogisticResult(10,2, 0.5)
GetLogisticResult(100,2, 0.5)
GetLogisticResult(1000,2, 0.5)

GetLogisticResult(10,2, 0.9)
GetLogisticResult(100,2, 0.9)
GetLogisticResult(1000,2, 0.9)
#mechanism 2: Bernoulli
GetLogisticResult_B(10,2, 0.1)
GetLogisticResult_B(100,2, 0.1)
GetLogisticResult_B(1000,2, 0.1)

GetLogisticResult_B(10,2, 0.5)
GetLogisticResult_B(100,2, 0.5)
GetLogisticResult_B(1000,2, 0.5)


GetLogisticResult_B(10,2, 0.9)

GetLogisticResult_B(100,2, 0.9)

GetLogisticResult_B(1000,2, 0.9)


"""## **2. Nearest Neibour Model (KNN algorithm)**"""

from sklearn.neighbors import KNeighborsClassifier

def GetKNNResult(n, d, rho):

  X_n = GenerateData_D(n, d, rho)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget(n,X_n,beta)

  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

  test_Y = GenerateTarget(1000,test_X, beta)

  KNNclassifier = KNeighborsClassifier(n_neighbors=5)

  KNNclassifier.fit(X_n, Y_n)

  pred_y = KNNclassifier.predict(test_X)

  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))


def GetKNNResult_B(n, d, rho):

  X_n = GenerateData_D(n, d, rho)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget_B(n,X_n, beta)

  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

  test_Y = GenerateTarget_B(1000,test_X, beta)
```

```
    KNNclassifier = KNeighborsClassifier(n_neighbors=5)
    KNNclassifier.fit(X_n, Y_n)
    pred_y = KNNclassifier.predict(test_X)
    print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

#mechanism 1
GetKNNResult(10,2, 0.1)
GetKNNResult(100,2, 0.1)
GetKNNResult(1000,2, 0.1)

GetKNNResult(10,2, 0.5)
GetKNNResult(100,2, 0.5)
GetKNNResult(1000,2, 0.5)

GetKNNResult(10,2, 0.9)
GetKNNResult(100,2, 0.9)
GetKNNResult(1000,2, 0.9)
#mechanism 2:bernoulli
GetKNNResult_B(10,2, 0.1)
GetKNNResult_B(100,2, 0.1)
GetKNNResult_B(1000,2, 0.1)

GetKNNResult_B(10,2, 0.5)
GetKNNResult_B(100,2, 0.5)
GetKNNResult_B(1000,2, 0.5)
GetKNNResult_B(10,2, 0.9)

GetKNNResult_B(100,2, 0.9)

GetKNNResult_B(1000,2, 0.9)


"""## **3. Support Vector Machine (SVM) Model**"""

from sklearn.svm import SVC

def GetSVMResult(n, d, rho):

  X_n = GenerateData_D(n, d, rho)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget(n, X_n, beta)

  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

  test_Y = GenerateTarget(1000, test_X, beta)

  SVMclassifier = SVC(kernel='rbf', random_state = 1)

  SVMclassifier.fit(X_n,Y_n)

  pred_y = SVMclassifier.predict(test_X)

  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))


def GetSVMResult_B(n, d, rho):

  X_n = GenerateData_D(n, d, rho)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget_B(n, X_n, beta)

  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

  test_Y = GenerateTarget_B(1000, test_X, beta)

  SVMclassifier = SVC(kernel='rbf', random_state = 1)
```

```
    SVMclassifier.fit(X_n,Y_n)
    pred_y = SVMclassifier.predict(test_X)
    print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

#mechanism 1
GetSVMResult(10,2, 0.1)
GetSVMResult(100,2, 0.1)
GetSVMResult(1000,2, 0.1)

GetSVMResult(10,2, 0.5)
GetSVMResult(100,2, 0.5)
GetSVMResult(1000,2, 0.5)

GetSVMResult(10,2, 0.9)
GetSVMResult(100,2, 0.9)
GetSVMResult(1000,2, 0.9)
#mechanism 2: bernoulli
GetSVMResult_B(10,2, 0.1)
GetSVMResult_B(100,2, 0.1)
GetSVMResult_B(1000,2, 0.1)

GetSVMResult_B(10,2, 0.5)
GetSVMResult_B(100,2, 0.5)
GetSVMResult_B(1000,2, 0.5)
GetSVMResult_B(10,2, 0.9)

GetSVMResult_B(100,2, 0.9)

GetSVMResult_B(1000,2, 0.9)


"""## **4. Random Forest Model**"""

from sklearn.ensemble import RandomForestClassifier

def GetRandomForestResult(n, d, rho):

  X_n = GenerateData_D(n, d, rho)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget(n, X_n, beta)

  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

  test_Y = GenerateTarget(1000, test_X, beta)

  RFclassifier = RandomForestClassifier(n_estimators=100) #set the number of tre

  RFclassifier.fit(X_n,Y_n)

  pred_y = RFclassifier.predict(test_X)

  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))


def GetRandomForestResult_B(n, d, rho):

  X_n = GenerateData_D(n, d, rho)

  beta = np.random.dirichlet(np.ones(d),size=1)

  Y_n = GenerateTarget_B(n, X_n, beta)

  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

  test_Y = GenerateTarget_B(1000, test_X, beta)

  RFclassifier = RandomForestClassifier(n_estimators=100) #set the number of tre

  RFclassifier.fit(X_n,Y_n)
  pred_y = RFclassifier.predict(test_X)
  print('Accuracy: ',metrics.accuracy_score(test_Y, pred_y))

#mechanism 1
GetRandomForestResult(10,2, 0.1)
GetRandomForestResult(100,2, 0.1)
GetRandomForestResult(1000,2, 0.1)

GetRandomForestResult(10,2, 0.5)
GetRandomForestResult(100,2, 0.5)
GetRandomForestResult(1000,2, 0.5)

GetRandomForestResult(10,2, 0.9)
GetRandomForestResult(100,2, 0.9)
GetRandomForestResult(1000,2, 0.9)
#mechanism 2: bernoulli
GetRandomForestResult_B(10,2, 0.1)
GetRandomForestResult_B(100,2, 0.1)
GetRandomForestResult_B(1000,2, 0.1)

GetRandomForestResult_B(10,2, 0.5)
GetRandomForestResult_B(100,2, 0.5)
GetRandomForestResult_B(1000,2, 0.5)

GetRandomForestResult_B(10,2, 0.9)
GetRandomForestResult_B(100,2, 0.9)
GetRandomForestResult_B(1000,2, 0.9)


"""## **5. Feed-Forward Neural Network Model**"""
import keras
from keras.models import Sequential,Input,Model
from keras.layers import Conv2D
from keras.layers import Dense, Flatten, Activation
def GetFFNNResult(n, d, rho):
  X_n = GenerateData_D(n, d, rho)
  beta = np.random.dirichlet(np.ones(d),size=1)
  Y_n = GenerateTarget(n, X_n, beta)
  test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test
  test_Y = GenerateTarget(1000, test_X, beta)
  X_n = np.array(X_n)
  Y_n = np.array(Y_n)
  test_X = np.array(test_X)
  test_Y = np.array(test_Y)
  #Build the feedforward NN model
  NNmodel = Sequential()
  NNmodel.add(Dense(64, input_shape = (1,d), activation = "relu"))
#The first layer
  NNmodel.add(Dense(32, activation='relu'))   #The second layer
  NNmodel.add(Dense(1, activation='sigmoid'))      #Output layer
  #Compile the feed-forward NN model
```

```
    NNmodel.compile(loss = "binary_crossentropy", optimizer = "Adam",metrics=['acc
    NNmodel.fit(X_n, Y_n, epochs=100, batch_size=10, verbose=0)
    _, accuracy = NNmodel.evaluate(test_X, test_Y)
    print('Accuracy: %.2f' % (accuracy))


def GetFFNNResult_B(n, d, rho):
    X_n = GenerateData_D(n, d, rho)
    beta = np.random.dirichlet(np.ones(d),size=1)
    Y_n = GenerateTarget_B(n, X_n, beta)
    test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test
    test_Y = GenerateTarget_B(1000, test_X, beta)
    X_n = np.array(X_n)
    Y_n = np.array(Y_n)
    test_X = np.array(test_X)
    test_Y = np.array(test_Y)
    #Build the feedforward NN model
    NNmodel = Sequential()
    NNmodel.add(Dense(64, input_shape = (1,d), activation = "relu"))
#The first layer
    NNmodel.add(Dense(32, activation='relu'))    #The second layer
    NNmodel.add(Dense(1, activation='sigmoid'))      #Output layer
    #Compile the NN model
    NNmodel.compile(loss = "binary_crossentropy", optimizer = "Adam",metrics=['acc
    NNmodel.fit(X_n, Y_n, epochs=100, batch_size=10, verbose=0)
    _, accuracy = NNmodel.evaluate(test_X, test_Y)
    print('Accuracy: %.2f' % (accuracy))


#mechanism 1
GetFFNNResult(10,2, 0.1)
GetFFNNResult(100,2, 0.1)
GetFFNNResult(1000,2, 0.1)

GetFFNNResult(10,2, 0.5)
GetFFNNResult(100,2, 0.5)
GetFFNNResult(1000,2, 0.5)

GetFFNNResult(10,2, 0.9)
GetFFNNResult(100,2, 0.9)
GetFFNNResult(1000,2, 0.9)
#mechanism 2: bernoulli
GetFFNNResult_B(10,2, 0.1)
GetFFNNResult_B(100,2, 0.1)
GetFFNNResult_B(1000,2, 0.1)

GetFFNNResult_B(10,2, 0.5)
GetFFNNResult_B(100,2, 0.5)
GetFFNNResult_B(1000,2, 0.5)

GetFFNNResult_B(10,2, 0.9)
GetFFNNResult_B(100,2, 0.9)
GetFFNNResult_B(1000,2, 0.9)


"""## **6. XG-boost Model**"""

from xgboost import XGBClassifier

def GetXGboostResult(n, d, rho):

    X_n = GenerateData_D(n, d, rho)

    beta = np.random.dirichlet(np.ones(d),size=1)

    Y_n = GenerateTarget(n, X_n, beta)

    test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

    test_Y = GenerateTarget(1000, test_X, beta)

    XGboost_classifier = XGBClassifier()

    XGboost_classifier.fit(np.asarray(X_n),np.asarray(Y_n))

    pred_y = XGboost_classifier.predict(np.asarray(test_X))

    predictions = [round(value) for value in pred_y]

    print('Accuracy: ',metrics.accuracy_score(test_Y, predictions))


def GetXGboostResult_B(n, d, rho):

    X_n = GenerateData_D(n, d, rho)

    beta = np.random.dirichlet(np.ones(d),size=1)

    Y_n = GenerateTarget_B(n, X_n, beta)

    test_X = GenerateData_D(1000,d, rho) #to get accurate results in accuracy test

    test_Y = GenerateTarget_B(1000, test_X, beta)

    XGboost_classifier = XGBClassifier()

    XGboost_classifier.fit(np.asarray(X_n),np.asarray(Y_n))

    pred_y = XGboost_classifier.predict(np.asarray(test_X))
```

```
    predictions = [round(value) for value in pred_y]
    print('Accuracy:␣',metrics.accuracy_score(test_Y, predictions))


#mechanism 1
GetXGboostResult(10,2, 0.1)
GetXGboostResult(100,2, 0.1)
GetXGboostResult(1000,2, 0.1)

GetXGboostResult(10,2, 0.5)
GetXGboostResult(100,2, 0.5)
GetXGboostResult(1000,2, 0.5)

GetXGboostResult(10,2, 0.9)
GetXGboostResult(100,2, 0.9)
GetXGboostResult(1000,2, 0.9)
#mechanism 2: bernoulli
GetXGboostResult_B(10,2, 0.1)
GetXGboostResult_B(100,2, 0.1)
GetXGboostResult_B(1000,2, 0.1)

GetXGboostResult_B(10,2, 0.5)
GetXGboostResult_B(100,2, 0.5)
GetXGboostResult_B(1000,2, 0.5)

GetXGboostResult_B(10,2, 0.9)
GetXGboostResult_B(100,2, 0.9)
GetXGboostResult_B(1000,2, 0.9)
```