

---

# CS7304H: Statistical Learning

## Final Project Report

---

**Hongjie Fang**

(Student No. 022033910104)

Department of Computer Science and Engineering  
Shanghai Jiao Tong University  
galaxies@sjtu.edu.cn

## 1 Requirements

In this project, we are required to complete a 20-class classification task with noisy data. There are 6666 feature samples for training and 2857 feature samples for testing. The training samples is kept clean, while the testing samples is noisy.

## 2 Methods

### 2.1 Preliminaries

**K Nearest Neighbors (KNN)** In statistical learning, K Nearest Neighbors (*abbrev.* KNN) [1, 2] is a non-parametric supervised learning method, which can be used for both classification and regression. In classification, we extract  $k$  closest training examples of the testing sample into a small dataset, then the testing sample is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its  $k$  nearest neighbors.

**Support Vector Machine (SVM)** In statistical learning, Support Vector Machine (*abbrev.* SVM) [3] is a supervised learning method with associated learning algorithms that analyzes data for classification and regression analysis. Classic SVM is a binary classification methods with response  $y_i = \pm 1$ , and it can be turned into an optimization problem as follows.

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|^2 \\ \text{s.t.} \quad & \forall i, \gamma_i = y_i(w^\top X_i + b) \geq 1 \end{aligned} \tag{1}$$

where  $X, y$  are training samples,  $w, b$  are parameters of SVM and  $\gamma$  is the functional margin of each sample. Using Lagrange multipliers, we can construct its loss function as follows.

$$L(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i [y_i(w^\top X_i + b) - 1] \tag{2}$$

Then, we can use the Karush-Kuhn-Tucker conditions (*abbrev.* KKT conditions) [4] to optimize the objective iteratively. Moreover, we can apply kernel methods to SVM by defining kernels  $K(\cdot, \cdot)$  and only focusing in  $K(X_i, X_j)$  instead of  $X_i^\top X_j$  in the original SVM. The kernel should satisfy the Mercer theorem [5], which states that the kernel should be symmetric and positive semi-definite.

**Multi-Layer Perceptron (MLP)** In machine learning, a Multi-Layer Perceptron (*abbrev.* MLP) is the feed-forward artificial neural networks with fully connected layers and non-linear activations). MLPs are sometimes colloquially referred to as vanilla neural networks, especially when they have a single hidden layer [6]. An MLP consists of at least 3 layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called back-propagation [7] for training.

## 2.2 KNN: with Distance Definition

In KNN, besides the hyper-parameter  $k$ , another important setting is the definition of the distance. KNN uses the Euclidean distance by default. Some other distance definitions are listed below.

- **Minkowski Distance** (*a.k.a.*  $L_p$  distance). The Minkowski distance is defined as

$$d_p(\mathbf{x}, \mathbf{y}) = \left( \sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}} \quad (3)$$

for features  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  and  $\mathbf{y} = (y_1, y_2, \dots, y_n)$ , and  $p$  is the hyper-parameter. The  $p = 0$  case is known as **Hamming distance**, the  $p = 1$  case is known as the **Manhattan distance**, and the  $p = 2$  case is known as the **Euclidean distance**.

- **Cosine Distance**. The cosine distance is defined as

$$d_C(\mathbf{x}, \mathbf{y}) = 1 - \cos \langle \mathbf{x}, \mathbf{y} \rangle = 1 - \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \quad (4)$$

for features  $\mathbf{x}$  and  $\mathbf{y}$ .

## 2.3 SVM: from Binary to Categorical

Ordinary SVM usually solves binary classification problems, but when it comes to categorical classification problems with  $C$  classes  $1, 2, \dots, C$ , we should make a few modifications to let it adapt to the new settings. Generally, there are two methods to extend ordinary binary SVM to categorical SVM, listed as follows.

- **One-versus-One** (*abbrev.*, OvO). We enumerate all class pairs  $(x, y)$  where  $x \neq y$  and  $x, y = 1, 2, \dots, C$ . For each class pair, we use a binary SVM classifier to classify them. Therefore, there are totally  $C \times (C - 1)/2$  binary SVM classifiers and the results forms an  $C \times C$  matrix without the diagonal elements. The final category results are obtained by voting according to the results in the matrix. In a summary, there are  $O(C^2)$  classifiers and each classifier only uses sample data from two classes in training process, thus results in sample inefficiency and time inefficiency.
- **One-versus-Rest** (*abbrev.*, OvR). For every class, we use a binary SVM classifier to classify whether a sample belongs to this class. If there are multiple classifiers that gives the positive results, then we choose the class that has the highest confidence score. Therefore, there are only  $C$  binary SVM classifiers, and every samples can be used by every classifiers. Hence, OvR method is more sample efficient and time efficient than OvO method.

There are also other methods like Many-versus-Many (*abbrev.*, MvM), which are much more complicated, and is less appropriate for categorical SVM settings. Hence, we only consider the previous two methods (OvO and OvR) in categorical SVM.

## 2.4 SVM: with Kernels

SVM can efficiently and effectively classify samples when they are linearly divisible. Consider the circumstances where samples are not fully linear divisible or even not linear divisible at all, using ordinary SVM might lead to worse performance. However, we can use kernel-based SVMs under these situations. The kernels can map the features space into a high-dimensional space defined by the kernels, where the samples might be linearly divisible, and resulting in better performance.

Formally, let  $K : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$  be given as the kernel function, then we replace every  $A^\top B$  with  $K(A, B) = \phi(A)^\top \phi(B)$  in SVM derivation, where  $A, B \in \{X_1, X_2, \dots, X_N, x\}$ . Then,  $K$  is a valid (Mercer) kernel [5] if and only if

- The kernel matrix is symmetric, *i.e.*,  $K(A, B) = \phi(A)^\top \phi(B) = \phi(B)^\top \phi(A) = K(B, A)$ .
- The kernel matrix is positive semi-definite, *i.e.*, if we define the kernel matrix  $K$  as an  $N \times N$  matrix where  $K_{ij} = K(X_i, X_j)$ , then for arbitrary vector  $z$ , we have  $z^\top K z \geq 0$ . Only under this condition, we can decompose  $K(A, B)$  as  $\phi(A)^\top \phi(B)$ .

Here are some commonly used (Mercer) kernels listed below.

- **Linear Kernel.** The linear kernel shown as follows is the ordinary kernel we used in discussions about SVM.

$$K(A, B) = A^\top B \quad (5)$$

- **Radial Base Function Kernel** (*a.k.a.* RBF kernel, Gaussian kernel). The RBF kernel is defined as

$$K(A, B) = \exp\left(-\frac{\|A - B\|^2}{2\sigma^2}\right) = \exp\left(-\gamma \|A - B\|^2\right) \quad (6)$$

where  $\sigma > 0$  is a hyper-parameter, and  $\gamma = 1/2\sigma^2$ . The transformation  $\phi$  for RBF kernel actually maps the initial feature to an infinite-dimensional space, since we can interpret  $K(A, B)$  using Taylor's expansion as follows.

$$\begin{aligned} K(A, B) &= \exp\left(-\frac{\|A - B\|^2}{2\sigma^2}\right) \\ &= \exp\left(-\frac{\|A\|^2 + \|B\|^2 - 2\langle A, B \rangle}{2\sigma^2}\right) \\ &= \exp\left(-\frac{\|A\|^2 + \|B\|^2}{2\sigma^2}\right) \exp\left(\frac{\langle A, B \rangle}{\sigma^2}\right) \\ &= \exp\left(-\frac{\|A\|^2 + \|B\|^2}{2\sigma^2}\right) \sum_{k=0}^{\infty} \frac{\langle A, B \rangle^k}{\sigma^{2k} k!} \end{aligned} \quad (7)$$

where  $\langle A, B \rangle^k$  can enumerate every  $k$ -ordered polynomials formed by  $A$  and  $B$ . Since we enumerate  $k$  from 0 to infinite, we can get all polynomials formed by  $A, B$ , which means that the transformation function is able to map  $X$  into an infinite space  $1, X, X^2, X^3, \dots$ .

- **Sigmoid Kernel.** The sigmoid kernel is defined as

$$K(A, B) = \tanh(aA^\top B + c) \quad (8)$$

where  $\tanh(x) = (1 - e^{-2x})/(1 + e^{-2x})$ , and  $a, c$  are hyper-parameters.

- **Polynomial Kernel.** The polynomial kernel is defined as

$$K(A, B) = (aA^\top B + c)^d \quad (9)$$

where  $d$  is the degree hyper-parameters, and  $a, c$  are hyper-parameters. Similar to the explanation in RBF kernel, using binomial expansion method, we can come to the conclusion that the transformation function of polynomial kernel actually maps  $X$  into a higher-dimensional space  $1, X, X^2, \dots, X^d$ .

## 2.5 MLP: Classification, Dropout Layer and Regularizations

**Classifications** MLP are initially designed for regression problems, but it can easily extend to classification tasks. For classification tasks with  $C$  classes, the output layer can outputs  $C$  values representing the confidence score of the sample belonging to each class. Then, we use softmax layer to normalize the confidence scores to the probability distribution, *i.e.*,

$$p_i = \frac{\exp s_i}{\sum_{j=1}^C \exp s_j} \quad (10)$$

Therefore, we can use **cross-entropy** loss between the distribution  $\{p_i\}$  and the one-hot target distribution  $p^*$  for classification tasks.

**Dropout** To improve the generalization ability of vanilla MLP, several techniques like Dropout layer [8] have been proposed in the past decade. Dropout methods ignore a certain proportion of neurons in MLP at a time, and only use the rest of neurons in calculations. Using Dropout layer in neural networks like MLP can effectively prevent it from over-fitting and poor generalization ability.

**Regularization** Another approach that can prevent MLP from overfitting is the regularization term. If we denote our MLP network as  $f_\theta$ , then a regularization term  $\lambda \|\theta\|^p$  should be added to the original loss function, where  $\lambda$  is a hyper-parameters. The case when  $p = 1$  is called Lasso term and the case when  $p = 2$  is called ridge term. Numerical experiments have demonstrated the effect of the regularization terms in improving the generalization ability.

## 2.6 Model Selection

In this section, we introduce two model selection methods used in this project, namely holdout method and cross validation. There are also other model selection methods like bootstrap methods, and they can be used in complex tasks. In this simple task the holdout method and cross validation are enough for model selection.

**Holdout Method** Holdout method essentially divides the whole dataset for training into two parts: training set and validation set. There are several methods for splitting the two parts, and the most simple one is the **random sampling**, *i.e.*, randomly split a set of samples out of the whole samples as the validation set, and the rest samples form the training set. However, in the case of unbalanced labels or small dataset, the label distribution in the randomly divided validation set could be inconsistent with the distribution in the whole dataset. This sample skewing problem may leads to ineffective and inaccurate validation process. Another sampling method called **stratified sampling** can be applied to solve the problem. The stratified sampling first divides the whole samples into several parts according to its labels, and calculates the label distribution of the dataset. Then it randomly sample several samples inside each parts and combine them together to form the validation set. The rest samples form the training set. The illustration of random sampling and stratified sampling are shown in Fig. 1.

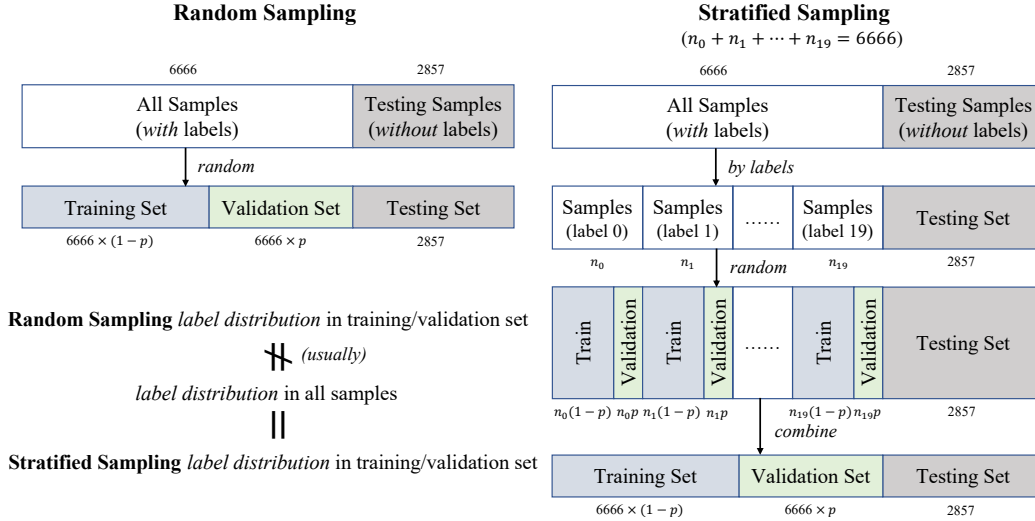


Figure 1: The Illustration of Holdout Method

**Cross Validation** Cross validation ( **$K$ -fold cross validation**) method divides the whole dataset into  $K$  parts, and trains the models for  $K$  times. Every time we take one part out of the dataset for validation, and use the rest parts for training. The final validation results is the average results during the  $K$  validations. If  $K = N$  then the cross validation is also called **leave-one-out cross validation**. Usually,  $K$  represents the trade-off between bias and variance. The larger the  $K$  is, more data is put into training, which results in less bias. However, the variance of the model might increases since there are correlations of data between each time. Empirically, we usually choose  $K = 5$  or  $K = 10$  for balance of bias and variance.

## 2.7 Dealing with Noise

Since random noise is added into the testing samples, we need to deal with the noise in testing features. Assume that the feature without noise follow Gaussian distribution  $\mathcal{N}(\mu_X, \sigma_X^2)$ , and the noise is the additive Gaussian noise  $\epsilon \sim \mathcal{N}(\mu_\epsilon, \sigma_\epsilon^2)$ , then the noisy features  $X' = X + \epsilon$  should follow Gaussian distribution  $\mathcal{N}(\mu_X + \mu_\epsilon, \sigma_X^2 + \sigma_\epsilon^2) \triangleq \mathcal{N}(\mu_{X'}, \sigma_{X'}^2)$ . Therefore, we can calculate the statistics  $\mu$  and  $\sigma$  from the clean training features  $X$  and noisy testing features  $X'$  respectively, and then we can compute the parameters  $\mu_\epsilon$  and  $\sigma_\epsilon$  of the additive Gaussian noise  $\epsilon$ .

After calculating the parameters of the additive Gaussian noise, we are now able to describe the noise more appropriately. Then, we introduce several ways to deal with the noise as follows.

- **Add Noise to Validation Set.** To adapt with the noisy features in testing set, we can manually add the same noise into the validation set. Therefore, the validation accuracy can reflect the testing accuracy more accurately, since the features in both sets are well aligned.
- **Apply Filters to Testing Set.** We can also denoise the noisy testing features to improve the performance. By our assumptions of additive Gaussian noise, we can use Gaussian filters to denoise the noisy features. Thanks to the parameters we calculated, we are now able to directly apply them to the filter. However, the filter might smoother the value of the features, resulting in poor performance. Therefore we do not consider this method in the experiments.
- **Generate Additional Noisy Data for Training.** To enhance the ability of the models, we can also generate additional noisy data using the noise parameters calculated before. Therefore, the models might be better adapted to testing features with noise. However, it would be better to keep the training data clean, therefore we do not consider this method in the experiments.

Therefore, in the experiments, we add noise to the validation set and perform model selection based on this noisy validation set.

## 3 Experiments

### 3.1 Data Processing

**Data Statistics** We summarize the statistics of data in Tab. 1. From the statistics we can see that the features are well normalized, thus we do not need to perform normalizations during experiments. Moreover, the noisy testing features has higher standard deviation than the training and validation features without noise. We also summarize the statistics of each label (class) in data in Tab. 2. From the statistics we can observe that the data is relatively balanced for each class.

Table 1: The Statistics of Training and Testing Features

Features	Noisy	Avg.	Std.	Min.	Max.	Median
Training + Validation	✗	$-4.11 \times 10^{-4}$	0.399	-4.23	12.43	$-3.38 \times 10^{-3}$
Testing	✓	$-9.22 \times 10^{-4}$	0.892	-4.89	14.04	$-3.13 \times 10^{-3}$

Table 2: The Statistics of Training Labels

Label	Count	Label	Count	Label	Count	Label	Count	Label	Count
0	375	1	366	2	316	3	325	4	328
5	336	6	343	7	315	8	336	9	319
10	325	11	341	12	342	13	339	14	318
15	329	16	324	17	336	18	327	19	326

**Noise** Follow the descriptions in Section 2.7, we plug the values of  $\mu_X, \mu_{X'}, \sigma_X$  and  $\sigma_{X'}$  in Tab. 1 into the formula, and then we can arrive at  $\mu_\epsilon = -5.11 \times 10^{-4} \approx 0$  and  $\sigma_\epsilon = 0.798 \approx 0.8$ , where the approximations are performed under the consideration of the feature scale shown in statistics part. Therefore, **under the previous hypotheses, we may conclude that the noise  $\epsilon$  follows Gaussian distribution  $\mathcal{N}(0, 0.8)$ .**

**Holdout Method** We manually implement random sampling and stratified sampling method in `process/holdout.py`. For all methods, we set splitting parameter  $p = 0.1$ , that is, 90% of the data is used for training and the rest is used for validation. For random sampling method, there are 5999 training samples and 667 validation samples. For stratified sampling method, there are 5991 training samples and 675 validation samples. We denote holdout method with random sampling as “H(r)”, and denote holdout method with stratified sampling as “H(s)” in evaluation tables.

**Cross Validation** We use  $K$ -fold cross validation and set  $K = 5$  empirically. We manually implement the cross validation method using stratified sampling in `process/cross-validation.py` and `dataset/splitter.py`. As a result, each part has 1326, 1326, 1326, 1326, 1362 samples respectively. We denote cross validation with stratified sampling as “CV(s)” in evaluation tables.

### 3.2 Implementations

**KNN** We use numpy library to implement KNN from scratch. Our implemented KNN supports two types of distance introduced in Section 2.2: Minkowski distance (with order  $p$ ) and cosine distance.

**SVM** We use numpy library to implement SVM from scratch. Our implemented SVM supports kernel methods introduced in Section 2.4, and use OvR method to support categorical classification due to the inefficiency of OvO method. However, our implemented SVM is less efficient than the SVM in the `sklearn` library, since Python runs much slower than basic languages like C and C++, in which the `libsvm` library inside the `sklearn` library implements the SVM. Therefore we use the SVM in `sklearn` library in experiments.

**MLP** We use `torch` library to implement MLP from scratch. The architectures of our implemented MLPs will be discussed later. We use AdamW optimizers [9] and a learning rate of  $10^{-3}$ .

### 3.3 Evaluation Results

**KNN** We evaluate our KNN models with different hyper-parameter settings (distance type and  $k$ ) using holdout method and cross-validation method respectively. We also use the “adding noise to validation set” trick during evaluation, as discussed in Section 2.7. The noise added into the validation set is fixed during all experiments for fair of comparisons. The results are shown in Tab. 3.

We summarize the model selection methods and their selected models under every circumstances (w/wo noise in validation set) in Tab. 4. We also test the performance of the selected models on testing set in Kaggle platform, and the results are also included in the table and Fig. 2.

From Tab. 3 and Tab. 4 we observe that the validation accuracy reflected by stratified sampling is stabler than the validation accuracy reflected by the random sampling, though holdout method with random sampling selects the model with highest testing accuracy. **Cross validation is not only stabler than holdout methods, but it also selects the relative optimal model with the second highest testing accuracy, showing its advantage over holdout methods.**

We also observe that **adding noise in validation set can enhance the model selection methods, making validation accuracy reflect the testing accuracy, and then improve model selection methods further.**

	result_k_50_cosine.csv Complete · now	0.92647	<input type="checkbox"/>
	result_k_20_cosine.csv Complete · 1s ago	0.92997	<input type="checkbox"/>
	result_k_10_cosine.csv Complete · 7m ago	0.92927	<input type="checkbox"/>

Figure 2: The Evaluation Results of KNNs on Testing Set in Kaggle Platform

In conclusion, our selected KNN models is the KNN with cosine distance and  $k = 20$ .

Table 3: The Evaluation Results of KNNs using Different Model Selection Methods

Distance Type	$k$	Validation Accuracy					
		without noise in validation			with noise in validation		
		H(r)	H(s)	CV(s)	H(r)	H(s)	CV(s)
$L_0$	5	5.25%	5.33%	5.23%	4.20%	5.63%	5.36%
	10	6.15%	5.78%	5.10%	5.70%	5.63%	5.21%
	20	4.20%	5.04%	5.28%	4.20%	4.74%	5.10%
	50	5.70%	5.33%	5.43%	5.70%	4.89%	4.92%
	100	5.70%	5.04%	4.95%	5.70%	5.04%	5.05%
$L_1$	5	93.10%	93.33%	93.16%	91.30%	92.30%	91.81%
	10	93.10%	93.33%	93.18%	91.90%	92.44%	91.84%
	20	93.55%	92.74%	93.19%	91.30%	91.41%	91.69%
	50	93.10%	92.44%	92.88%	91.00%	91.11%	91.44%
	100	92.65%	92.30%	92.31%	90.25%	90.52%	90.70%
$L_2$	5	94.15%	93.48%	93.48%	92.65%	92.15%	92.17%
	10	93.70%	93.48%	93.37%	92.05%	92.15%	92.25%
	20	94.00%	93.19%	93.16%	91.75%	92.00%	92.06%
	50	93.25%	92.89%	92.89%	91.00%	91.56%	91.90%
	100	92.50%	92.44%	92.43%	89.96%	91.11%	91.30%
Cosine	5	93.55%	93.48%	93.31%	92.50%	92.00%	92.38%
	10	<b>94.45%</b>	92.89%	<b>93.72%</b>	92.50%	92.29%	<b>92.92%</b>
	20	93.70%	93.33%	93.55%	<b>92.95%</b>	92.59%	92.71%
	50	93.10%	<b>93.63%</b>	93.63%	91.90%	<b>93.48%</b>	92.43%
	100	92.65%	92.74%	93.21%	91.45%	92.59%	92.20%

Table 4: The Model Selection Results of KNNs and Their Evaluation Results on Testing Set

w/wo Noise	Model Selection Method	Selected Model	Testing Accuracy (Rank)
without	Holdout (random)	Cosine, $k = 10$	92.93% (2)
	Holdout (stratified)	Cosine, $k = 50$	92.65% (3)
	Cross-Validation (stratified)	Cosine, $k = 10$	92.93% (2)
with	Holdout (random)	Cosine, $k = 20$	<b>93.00%</b> (1)
	Holdout (stratified)	Cosine, $k = 50$	92.65% (3)
	Cross-Validation (stratified)	Cosine, $k = 10$	92.93% (2)

**SVM** We evaluate our SVM models with different hyper-parameter settings (kernel type and penalty factor  $C$ ) using holdout method and cross-validation method respectively. Our categorical SVM models are implemented using OvR method due to the inefficiency of OvO method, as introduced before. We also use the “adding noise to validation set” trick during evaluation, as discussed in Section 2.7. The noise added into the validation set is fixed during all experiments for fair of comparisons. The results are shown in Tab. 5.

We summarize the model selection methods and their selected models under every circumstances (w/wo noise in validation set) in Tab. 6. We also test the performance of the selected models on testing set in Kaggle platform, and the results are also included in the table and Fig. 3.

From Tab. 5 and Tab. 6 we observe that the validation accuracy reflected by stratified sampling is stabler than the validation accuracy reflected by the random sampling, though holdout method with random sampling selects the model with highest testing accuracy. **Cross validation is not only stabler than holdout methods, but it also selects the optimal model with the highest testing accuracy when adding noise to validation set, showing its advantage over holdout methods.**

Moreover, it is worth noticing that SVMs with RBF kernel performs relatively well when using the original validation set, but it performs very bad when we add noise in the validation set, which aligns with its performance on testing set on Kaggle platform ( $\approx 6\%$ ). This suggests that **our “adding noise to validation set” is indeed helpful in model selection when dealing with noisy testing data.**

In conclusion, our selected SVM models is the SVM with polynomial kernels of  $d = 3$  and  $C = 0.5$ .

Table 5: The Evaluation Results of SVMs using Different Model Selection Methods

Kernel Type	$C$	Validation Accuracy					
		without noise in validation			with noise in validation		
		H(r)	H(s)	CV(s)	H(r)	H(s)	CV(s)
Linear	0.1	<b>95.20%</b>	<b>96.00%</b>	<b>95.85%</b>	91.75%	92.89%	92.23%
	0.5	<b>95.20%</b>	<b>96.00%</b>	<b>95.85%</b>	91.75%	92.89%	92.23%
	1.0	<b>95.20%</b>	<b>96.00%</b>	<b>95.85%</b>	91.75%	92.89%	92.23%
	5.0	<b>95.20%</b>	<b>96.00%</b>	<b>95.85%</b>	91.75%	92.89%	92.23%
RBF	0.1	89.51%	90.81%	89.74%	7.35%	4.89%	5.98%
	0.5	93.40%	94.07%	93.64%	5.55%	4.89%	5.33%
	1.0	94.30%	94.52%	94.95%	5.55%	6.07%	5.94%
	5.0	94.75%	94.67%	95.09%	5.55%	5.04%	8.14%
Sigmoid	0.1	93.70%	93.78%	94.35%	92.20%	91.85%	92.83%
	0.5	94.75%	95.26%	95.28%	92.80%	93.19%	93.12%
	1.0	<b>95.20%</b>	95.55%	95.40%	91.90%	93.33%	92.68%
	5.0	93.85%	94.67%	94.63%	87.56%	85.48%	87.21%
Poly ( $d = 3$ )	0.1	94.00%	93.63%	93.75%	92.05%	91.41%	91.82%
	0.5	94.90%	94.81%	95.10%	92.95%	92.89%	<b>93.55%</b>
	1.0	<b>95.20%</b>	95.41%	95.22%	<b>93.10%</b>	93.03%	93.49%
	5.0	94.90%	95.26%	95.00%	92.95%	<b>93.63%</b>	93.15%
Poly ( $d = 5$ )	0.1	52.77%	51.56%	48.21%	53.37%	53.04%	48.59%
	0.5	85.61%	85.33%	85.13%	82.76%	82.52%	82.34%
	1.0	90.40%	89.78%	90.56%	87.41%	86.81%	87.34%
	5.0	92.95%	92.59%	93.21%	90.70%	90.22%	90.71%

Table 6: The Model Selection Results of SVMs and Their Evaluation Results on Testing Set

w/wo Noise	Model Selection Method	Selected Model	Testing Accuracy (Rank)
without	Holdout (random)	Linear, $C = \text{any}$	93.28% (5)
	Holdout (random)	Sigmoid, $C = 1.0$	93.35% (4)
	Holdout (random)	Poly ( $d = 3$ ), $C = 1.0$	94.61% (2)
	Holdout (stratified)	Linear, $C = \text{any}$	93.28% (5)
	Cross-Validation (stratified)	Linear, $C = \text{any}$	93.28% (5)
with	Holdout (random)	Poly ( $d = 3$ ), $C = 1.0$	94.61% (2)
	Holdout (stratified)	Poly ( $d = 3$ ), $C = 5.0$	94.26% (3)
	Cross-Validation (stratified)	Poly ( $d = 3$ ), $C = 0.5$	<b>94.68% (1)</b>






	<b>result_poly_d_3_C_0.5.csv</b> Complete · 3m ago	<b>0.94677</b>	<input type="checkbox"/>
	<b>result_poly_d_3_C_5.0.csv</b> Complete · 8h ago	<b>0.94257</b>	<input type="checkbox"/>
	<b>result_poly_d_3_C_1.0.csv</b> Complete · 8h ago	<b>0.94607</b>	<input type="checkbox"/>
	<b>result_linear_C_0.5.csv</b> Complete · 8h ago	<b>0.93277</b>	<input type="checkbox"/>
	<b>result_sigmoid_C_1.0.csv</b> Complete · 8h ago	<b>0.93347</b>	<input type="checkbox"/>

Figure 3: The Evaluation Results of SVMs on Testing Set in Kaggle Platform



**MLP** We evaluate our MLP models with different hyper-parameter settings (architectures, regularizations, batch size) using holdout method (since cross-validation method is not supported in deep learning). The architecture of our implemented MLPs is shown in Fig. 4, where the dropout layer is optional. We also use the “adding noise to validation set” trick during evaluation, as discussed in Section 2.7. The noise added into the validation set is fixed during all experiments for fair of comparisons. The results are shown in Tab. 7.

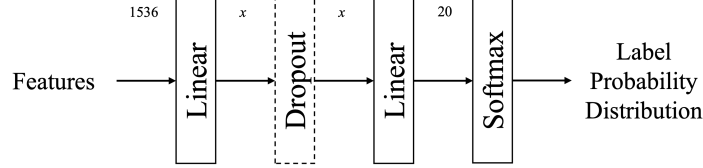


Figure 4: The Architecture of our implemented MLPs (1536,  $x$ , 20)

Table 7: The Evaluation Results of MLPs using Different Model Selection Methods

Architecture	w/wo Dropout	Reg.	Validation Accuracy			
			w. noise in val.		w. noise in val.	
			H(r)	H(s)	H(r)	H(s)
(1536, 32, 20)	without	no	96.10%	96.15%	90.70%	92.30%
	without	ridge	95.65%	96.00%	91.45%	92.15%
	with	no	95.80%	96.30%	92.20%	92.59%
	with	ridge	95.80%	96.00%	91.90%	<b>93.33%</b>
(1536, 64, 20)	without	no	95.65%	95.70%	<b>92.80%</b>	92.15%
	without	ridge	95.65%	<b>96.89%</b>	91.90%	92.89%
	with	no	95.65%	96.15%	92.35%	<b>93.33%</b>
	with	ridge	<b>96.25%</b>	96.15%	92.05%	92.59%
(1536, 128, 20)	without	no	94.90%	96.15%	91.30%	92.74%
	without	ridge	95.35%	95.85%	<b>92.80%</b>	92.44%
	with	no	96.10%	95.85%	91.15%	92.59%
	with	ridge	95.65%	96.15%	91.75%	92.30%

From Tab. 7 and Tab. 8 we observe that **the validation accuracy reflected by stratified sampling is stabler than the validation accuracy reflected by the random sampling, though holdout method with random sampling selects the model with highest testing accuracy.**

Also, we can find out that **adding dropout layer and regularization term indeed improve the performances when validating on noisy validation set**, which means these tricks enhance the generalization ability of the MLP model.

Moreover, from Tab. 8 we can find out that **our “adding noise to validation set” is indeed helpful in model selection when dealing with noisy testing data.**

Table 8: The Model Selection Results of MLPs and Their Evaluation Results on Testing Set

w/wo Noise	Model Selection Method	Selected Model	Testing Accuracy (Rank)
without	Holdout (random)	(1536, 64, 20), w. Dropout, ridge	91.63% (5)
	Holdout (stratified)	(1536, 64, 20), wo. Dropout, ridge	91.60% (6)
with	Holdout (random)	(1536, 64, 20), wo. Dropout	<b>92.79%</b> (1)
	Holdout (random)	(1536, 128, 20), wo. Dropout, ridge	92.02% (4)
	Holdout (stratified)	(1536, 32, 20), w. Dropout, ridge	92.30% (2)
	Holdout (stratified)	(1536, 64, 20), w. Dropout	92.16% (3)

In conclusion, our selected MLP model is the (1536, 64, 20) MLP without dropout layer and regularization term.

✓	result_h_64_w_drop_w_reg.csv Complete · 6m ago	0.91526	<input type="checkbox"/>
✓	result_h_32_w_drop_w_reg.csv Complete · 9m ago	0.92296	<input type="checkbox"/>
✓	result_h_64_w_drop_wo_reg.csv Complete · 11m ago	0.92156	<input type="checkbox"/>
✓	result_h_64_wo_drop_w_reg.csv Complete · 14m ago	0.92016	<input type="checkbox"/>
✓	result_h_64_wo_drop_wo_reg.csv Complete · 15m ago	0.92787	<input type="checkbox"/>
✓	result_h_64_wo_drop_w_reg.csv Complete · 16m ago	0.91596	<input type="checkbox"/>

Figure 5: The Evaluation Results of MLPs on Testing Set in Kaggle Platform

### 3.4 Final Results

In previous sections, we consider three types of models: KNN (conventional method), SVM (conventional method) and MLP (deep learning), and we use two model selection methods, namely holdout method (with random sampling and stratified sampling) and cross-validation to choose the best model for each type of model. In conclusion, our selected KNN, SVM and MLP models and their evaluation performances on testing set in Kaggle platform are shown in Tab. 9.

Table 9: The Final Evaluation Results on Testing Set

Method	Testing Accuracy
KNN (cosine distance, $k = 20$ )	93.00%
SVM (polynomial kernel of $d = 3$ , $C = 0.5$ )	<b>94.68%</b>
MLP ((1536, 64, 20), wo. Dropout)	92.79%

The evaluation results (screenshots) for those methods are also shown in Fig. 2, Fig. 3 and Fig. 5.

## 4 Conclusion

In this project, we implement **3 statistical learning method: KNN** (conventional method), **SVM** (conventional method) **and MLP** (deep learning), and use **2 model selection methods: holdout method** (with random sampling and stratified sampling) **and cross-validation** ( $k$ -fold cross validation).

To deal with the noisy testing data, we first assume that noise follows Gaussian distribution, and then calculate the parameters of the Gaussian distribution of the noise using the statistics of the features. Hence, we propose to **add noise of the same distribution in the validation set** to make the validation set more closer to the testing set, and the validation accuracy can better reflect the testing accuracy. In experiments, we find out that **our strategy improves the performance of model selection, and is indeed helpful in model selection when dealing with noisy testing data.**

**The final best testing accuracy on testing set in Kaggle platform is 94.68%**, which is achieved by SVM with polynomial kernel of degree  $d = 3$  and hyper-parameter  $C = 0.5$ . Therefore, we can conclude that sometimes conventional methods might perform better than deep learning methods, and **we need to choose the most appropriate method according to the problem settings**, instead of using deep learning method under all circumstances.

From the implementation process and the experiments, I gain more insight into various statistical learning algorithms and model selection methods, including KNN, SVM and MLP, as well as holdout method and cross-validation method. Also, our experiments find out that dropout layers and regularization terms can enhance the generalization ability of the MLP models, which aligns well with the knowledge learned in the course. Overall, I think this project benefits me a lot.

## Acknowledgement

I would like to express my deepest gratitude to Prof. Liqing Zhang for the detailed introduction and explanation on statistical learning.

I would like to thank the teaching assistants for answering our questions and providing us with supports during the course.

## References

- [1] E. Fix and J. L. Hodges, “Discriminatory analysis. nonparametric discrimination: Consistency properties,” *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.
- [2] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.
- [3] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [4] H. Kuhn and A. Tucker, “Nonlinear programming in proceedings of 2nd berkeley symposium (pp. 481–492),” *Berkeley: University of California Press*, 1951.
- [5] J. Mercer, “Functions of positive and negative type and their connection with the theory of integral equations,” *Philos. Transactions Royal Soc*, vol. 209, pp. 4–415, 1909.
- [6] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, vol. 2. Springer, 2009.
- [7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
- [8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [9] I. Loshchilov and F. Hutter, “Fixing weight decay regularization in adam,” 2018.

## Appendix

### A Environment

Our environment settings is listed below.

- **Operating System.** MacOS Big Sur with CPU.
- **Python.** Python 3.8

### B Codebase

#### B.1 Preliminary

Our codebase relies on several Python packages, including: `numpy`, `tqdm`, `pytorch` and `sklearn`. Please install the packages before running our codes.

#### B.2 Code

All the source codes in this project are available at

<https://github.com/Galaxies99/CS7304H-Project>

#### B.3 Checkpoints and Results

Please download checkpoints and the results at

<https://pan.baidu.com/s/1USGhKaoGsZtNmcxz--R1QA> (Extraction Code: 83c3)

Choose `logs.zip`, download it, and then extract the zipped file into a folder `logs`, and put it in the root directory of our codebase.

Here are the correspondence of the results in Tab. 9 and the `csv` file.

- **KNN** (cosine,  $k = 20$ ): `logs/knn/result_k_20_cosine.csv`;
- **SVM** (poly,  $d = 3$ ,  $C = 0.5$ ): `logs/svm/result_poly_d_3_C_0.5.csv`;
- **MLP** ((1536, 64, 20), wo. dropout): `logs/mlp/result_h_64_wo_drop_wo_reg.csv`, and its corresponding checkpoint is `logs/mlp/h_64_wo_drop_wo_reg_3.pth`.

#### B.4 Data Processing

**Recommendation** Please download all data that we used at

<https://pan.baidu.com/s/1USGhKaoGsZtNmcxz--R1QA> (Extraction Code: 83c3)

Choose `data.zip`, download it, and then extract the zipped file into a folder `data`, and put it in the root directory of our codebase.

**Process the Data by Yourselfs** You can also use the scripts in `process` to process the data by yourselves, but notice that the processing scripts might lead to different results in different device at different time. Therefore, the final result might be a little different. Here are the detailed explanations.

- `process/noise.py`: Generate noise for validation set;
- `process/holdout.py`: Generate the train-validation split for holdout method;
- `process/cross-validation.py`: Divide the data into several parts, which are used in cross-validation.

## B.5 Configurations

We provide all the configuration files we used in the experiments in `configs` folder. You may choose the configuration you want in inference or training process.

## B.6 Inference (Optional)

For inference, use the following command:

```
python run.py (--data [Data Path])
               --model [Model Type]
               --cfg [Configuration File]
               --inference_only
               --ckpt [Checkpoint Path]
               --output [Output Path]
```

where

- [Data Path] (optional) is the path to the data, default is data;
- [Model Type] is one of knn, svm, svm-ours (our slow implementation) and mlp;
- [Configuration File] is the path to the configuration file;
- [Checkpoint Path] is the path to the checkpoint;
- [Output Path] is the output file path of the inference process.

## B.7 Training (Optional)

For training (optional, since we provide trained checkpoints and results), use the following command:

```
python run.py (--data [Data Path])
               --model [Model Type]
               --cfg [Configuration File]
               --train_only
               --val [Model Selection Method]
               --split [Split File]
               (--ckpt [Checkpoint Path])
               (--val_with_noise)
               (--noise_path [Noise Path])
```

where

- [Data Path] (optional) is the path to the data, default is data;
- [Model Type] is one of knn, svm, svm-ours (our slow implementation) and mlp;
- [Configuration File] is the path to the configuration file;
- [Model Selection Method] is one of holdout and cross-validation;
- [Split File] is the path to the split file of model selection methods, generated in data processing scripts, usually in data folder;
  - For holdout method with random sampling, use `holdout_random_split.npy`;
  - For holdout method with stratified sampling, use `holdout_stratified_split.npy`;
  - For cross validation method, use `cross_validation_split.npy`.
- [Checkpoint Path] (optional) is the path to the checkpoint, if not specified, then the training process won't save any checkpoints;
- The `val_with_noise` option (optional) controls whether to add noise in the validation set;
- [Noise Path] (optional) specifies the path of the generated noise, usually in data folder.
  - For holdout method with random sampling, use `validation_manual_noise.npy`;
  - For holdout method with stratified sampling, use `validation_stratified_manual_noise.npy`;
  - For cross validation method, use `all_manual_noise.npy`.