

---

# Report of CS7304H 2025 Fall Project

---

UNikeEN

(Student No 025033xxxxxx)

Department of Computer Science and Engineering  
Shanghai Jiao Tong University

## 1 Requirements

In this project, we are required to complete a 10-category classification task with noisy data. There are totally 15000 1024-dim feature embedding samples for training and 2000 samples for testing. The testing samples are noisy, while the training samples are kept clean.

## 2 Preliminaries

**K-Nearest Neighbors (KNN)** In statistical learning, K-Nearest Neighbors (KNN) is a non-parametric supervised learning method for both classification and regression task. Given a training set  $\{(x_i, y_i)\}_{i=1}^n$  and a query sample  $x$ , KNN first finds the index set  $\mathcal{N}_k(x)$  of the  $k$  closest training examples to  $x$ . For classification, KNN assigns  $x$  to the most frequent class among its neighbors:

$$\hat{y}(x) = \arg \max_{c \in \mathcal{C}} \sum_{i \in \mathcal{N}_k(x)} \mathbf{1}(y_i = c).$$

For regression, KNN predicts by averaging the neighbor responses:

$$\hat{f}(x) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x)} y_i.$$

**Nearest Class Mean (NCM)** Nearest Class Mean (NCM) is a prototype-based classifier. It summarizes each class by its centroid (sample mean) computed from the training data, and predicts the label of a query point by comparing it to these class centroids. Because NCM compresses each class into a single representative prototype, it is simple, fast at test time, and often serves as a strong baseline when class clusters are roughly compact in feature space.

**Support Vector Machine (SVM)** Support Vector Machine (SVM) is a supervised learning method for binary classification with labels  $y_i \in \{+1, -1\}$ . Its core idea is to find a separating hyperplane that maximizes the margin, i.e., the distance between the decision boundary and the closest training samples from both classes. In the linearly separable case, SVM can be formulated as the maximum-margin optimization problem:

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

The final classifier predicts by the sign of the decision function:

$$\hat{y}(x) = \text{sign}(w^\top x + b).$$

Only a small subset of training samples (which are called support vectors) determine the optimal hyperplane.

**Multi-Layer Perceptron (MLP)** Multi-Layer Perceptron (MLP) is a feed-forward neural network composed of fully connected layers and nonlinear activation functions. By stacking multiple layers, MLP can learn flexible nonlinear mappings from inputs to outputs, making it suitable for both classification and regression. MLPs are typically trained with back-propagation and gradient-based optimization to minimize a chosen loss function on labeled training data.

### 3 Methods

#### 3.1 Dimensionality Reduction

**Principal Component Analysis (PCA)** Principal Component Analysis (PCA) is an unsupervised linear dimensionality reduction method. It projects high-dimensional data onto a low-dimensional subspace that preserves the maximum variance of the data. Formally, PCA finds an orthonormal projection matrix  $V_q \in \mathbb{R}^{p \times q}$  by solving

$$V_q = \arg \max_{V^\top V = I_q} \text{tr}(V^\top \Sigma V),$$

where  $\Sigma$  is the sample covariance matrix. In our experiments, PCA is used to reduce the feature dimension to 9, which balances information preservation and computational efficiency.

**Linear Discriminant Analysis (LDA)** Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction method that exploits class label information. It aims to find a projection that maximizes between-class variance while minimizing within-class variance:

$$V = \arg \max_V \frac{\text{tr}(V^\top S_B V)}{\text{tr}(V^\top S_W V)},$$

where  $S_B$  and  $S_W$  denote the between-class and within-class scatter matrices, respectively. Since LDA can reduce the data to at most  $C - 1$  dimensions for  $C$  classes, in our case the feature dimension is reduced to 9 accordingly.

#### 3.2 KNN and NCM with Different Distance Metrics

**Distance Metrics** Both KNN and NCM rely on distance measurements in the feature space. We consider the following commonly used distance metrics:

- **Minkowski Distance ( $L_p$  distance):**

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

where  $p$  is a hyper-parameter. Special cases include Manhattan distance ( $p = 1$ ) and Euclidean distance ( $p = 2$ ).

- **Cosine Distance:**

$$d_c(x, y) = 1 - \frac{x^\top y}{\|x\| \|y\|},$$

which measures the angular difference between feature vectors and is scale-invariant.

#### 3.3 SVM for Multi-Class Classification

**One-versus-One (OvO)** The One-versus-One (OvO) strategy extends a binary SVM to a  $C$ -class classification task by constructing a binary classifier for every unordered pair of classes. For each pair  $(c_i, c_j)$  with  $c_i \neq c_j$ , we extract the subset of training data belonging to these two classes only, and assign binary labels  $y \in \{+1, -1\}$  according to the class membership. Therefore, the total number of OvO classifiers is

$$\frac{C(C-1)}{2}.$$

At test time, an input  $x$  is fed into all pairwise classifiers; each classifier produces a predicted class label from its own two-class set. A common aggregation rule is majority voting: each classifier casts

one vote for one of its two classes, and the final output is the class with the highest total votes. If ties occur, one may further break ties using decision values (margins) from the corresponding classifiers, or use a predefined tie-breaking order.

**One-versus-Rest (OvR)** The One-versus-Rest (OvR) strategy trains  $C$  binary SVM classifiers. For each class  $c \in \{1, \dots, C\}$ , we define a binary label

$$y_i^{(c)} = \begin{cases} +1, & y_i = c, \\ -1, & y_i \neq c, \end{cases}$$

and train a binary SVM to separate class  $c$  from all remaining classes grouped together. This yields exactly  $C$  classifiers, and each classifier is trained using the full dataset: positives are samples from class  $c$ , while negatives are samples from all other classes. At test time, we evaluate all  $C$  decision functions  $\{f_c(x)\}_{c=1}^C$  and select the predicted class as

$$\hat{y} = \arg \max_{c \in \{1, \dots, C\}} f_c(x),$$

where  $f_c(x)$  is the signed decision score (margin) for class  $c$ . In practice, OvR may include calibration or scaling of the decision values so that scores from different binary classifiers are comparable, especially when class sizes are highly imbalanced.

### 3.4 MLP Optimization

This procedure keeps unmasked samples unchanged while adding random perturbations to the masked ones. Operationally, the mask changes across epochs, so different subsets of samples are perturbed at different training iterations, and the same sample may be seen both in clean and perturbed forms over the whole training process.

**Dropout** To prevent co-adaptation of hidden units, we adopt *dropout* during training. Concretely, at each optimization step, dropout randomly deactivates a subset of neurons (or feature channels) with a fixed probability, and the network is trained with this randomly thinned architecture. In implementation, dropout is only enabled in training mode; at inference time, all units are used with the corresponding scaling handled by the framework. In our MLP setting, dropout is applied to the hidden layers so that the representation learned by the network does not rely excessively on any single activation pattern.

**Regularization** We further apply *regularization* to control model complexity by penalizing large parameters. In practice, we consider standard weight penalties such as  $\ell_2$  and  $\ell_1$  regularization:  $\ell_2$  encourages small but nonzero weights (a shrinkage effect), while  $\ell_1$  promotes sparse solutions by driving some weights exactly to zero. These two penalties are closely related to classical linear-model regularization:  $\ell_2$  regularization is analogous to *ridge regression*, and  $\ell_1$  regularization is analogous to *lasso regression*. In our MLP optimization, the regularization term is added to the training objective together with the empirical loss, and the corresponding strength is treated as a tunable hyper-parameter.

### 3.5 Model Selection

**Cross Validation** To select appropriate models and hyper-parameters, we employ *cross validation* (CV) to estimate the generalization performance. Given a labeled dataset, CV first partitions the data into  $K$  disjoint subsets (folds) of approximately equal size. For each fold  $k = 1, \dots, K$ , the model is trained on the remaining  $K - 1$  folds and evaluated on the held-out fold. The validation error is then averaged over all folds to obtain an estimate of the expected generalization error.

In practice, cross validation is used to compare different model configurations, such as the number of neighbors in KNN, the choice between PCA and LDA for dimensionality reduction, the regularization strength in SVM, or optimization-related hyper-parameters in MLP. The model or hyper-parameter setting that achieves the lowest average validation error is selected. After model selection, the chosen configuration is retrained on the full training set and evaluated once on the test set to report the final performance.

### 3.6 Dealing with Noise

**Data Augmentation** Data augmentation is a commonly used strategy to improve the robustness and generalization ability of learning models by artificially increasing the diversity of the training data. Instead of collecting new samples, augmentation modifies existing training data in a controlled manner, so that the model can be exposed to a wider range of input variations.

In this experiment, we adopt a simple yet effective augmentation scheme based on additive Gaussian noise. Specifically, during each training epoch, Gaussian noise is injected into a subset of the training samples. Formally, given a clean input feature vector  $x$ , the augmented sample is generated as

$$x' = x + \epsilon,$$

where  $\epsilon$  is drawn from a zero-mean Gaussian distribution with a predefined variance. The noise is applied selectively according to a binary mask, such that only part of the mini-batch is augmented while the remaining samples are kept unchanged.

This augmentation strategy aims to simulate realistic perturbations and noise that may appear in practical testing scenarios, where input features are often corrupted by measurement errors or environmental interference. By repeatedly observing noisy variants of the same samples during training, the model is encouraged to learn more stable and noise-insensitive representations. As a result, the trained model is better aligned with the noisy test distribution and is expected to achieve improved generalization performance under noisy conditions.

## 4 Implementation

### 4.1 Pipeline Design

This section describes the engineering implementation details of our experimental pipeline.

To efficiently evaluate multiple models and hyperparameter configurations within a single experimental framework, we design a unified Pipeline abstraction consisting of a base class and a set of derived classes. The base pipeline handles common procedures shared by all methods, including dataset splitting, data preprocessing (e.g., normalization and dimensionality reduction), and experiment orchestration. Each derived pipeline corresponds to a specific learning method and is required to implement the `train` and `predict` interfaces defined in the base class.

The `train` function performs model training given a training set, while the `predict` function takes an input feature matrix  $X \in \mathbb{R}^{N \times d}$  in the form of a `numpy.ndarray` and outputs the corresponding predictions. To support flexible experimental protocols, we further define two execution stages. The `stage1` function conducts model training under a single train-validation split or multiple splits for cross-validation, by repeatedly invoking the `train` function. The `stage2` function applies the trained model to the target test set and produces final predictions.

At the outer level, a unified `main` function accepts a list of configuration strings, each specifying a complete experimental setting. Each configuration string encodes the model type, model-specific hyperparameters, and dimensionality reduction strategy, using underscores and the `@` symbol as separators. For example:

- `m1p_512_256_dp0.1@pca_128` denotes a two-hidden-layer MLP with 512 and 256 neurons respectively, dropout probability 0.1, and PCA applied to reduce the input dimensionality to 128.
- `knn_30_cos@lda_9` denotes a KNN classifier with  $k = 30$ , cosine distance metric, and LDA applied to reduce the input dimensionality to 9.

For individual methods, the implementations are summarized as follows.

- **Dimensionality Reduction (PCA / LDA).** We implement PCA and LDA as introduced in Section 3.1 using the `sklearn` library.
- **KNN.** We implement the KNN classifier described in Section 3.2 using `sklearn`. Our implementation supports both Euclidean ( $\ell_2$ ) and cosine distance metrics.

- **NCM.** We implement the Nearest Class Mean (NCM) classifier described in Section 3.2 from scratch using `numpy`. It also supports both Euclidean and cosine distances.
- **SVM.** We initially implement an SVM from scratch using `numpy`. Our implementation supports kernel methods introduced in Section 3.3 and extends binary SVM to multi-class classification using the One-versus-Rest (OvR) strategy, due to the inefficiency of One-versus-One (OvO). However, since a pure Python implementation is significantly slower than the optimized C/C++ backend (`libsvm`) used in `sklearn`, we report experimental results using the `sklearn` SVM implementation.
- **MLP.** We implement the Multi-Layer Perceptron (MLP) described in Section 3.4 using `PyTorch`, with several training optimizations including dropout, gradient clipping, and label smoothing. Unless otherwise specified, we fix the learning rate to  $1.0 \times 10^{-3}$ , train for 100 epochs, and use a batch size of 256.

All experiments are conducted on a single NVIDIA RTX 5090Dv2 GPU.

## 5 Experiments

### 5.1 Evaluation Results

**KNN.** We evaluate the performance of KNN models under different hyper-parameter settings, including the number of neighbors  $k$ , distance metrics, and dimensionality reduction methods. All experiments are conducted using fixed cross set split. The validation accuracy is reported as the model selection criterion.

To investigate the effect of feature preprocessing, we consider the following settings:

- **No dimensionality reduction**, i.e., using the normalized 1024-dim features directly.
- **PCA**, reducing features to a lower-dimensional subspace (e.g., 128 or 256 dims).
- **LDA**, reducing features to 9 dimensions (i.e.,  $C - 1$  for a 10-class classification task).

All models are evaluated under identical data splits to ensure fair comparison. Table 1 summarizes the validation accuracy of KNN models with different

Table 1: Validation accuracy (%) of KNN with different distance metrics, neighborhood sizes  $k$ , and dimensionality reduction methods.

Distance	$k$	None	PCA-128	PCA-256	LDA-9
$L_2$	10	83.20	84.07	83.80	84.53
	20	83.60	83.93	84.20	84.93
	30	83.87	84.27	84.13	84.60
	50	83.27	82.93	83.53	84.80
	80	82.07	83.00	82.13	85.00
Cosine	10	83.07	84.00	83.00	84.33
	20	84.27	84.53	84.73	84.80
	30	84.47	83.40	83.67	84.73
	50	84.00	82.60	83.67	84.33
	80	83.07	82.67	82.73	84.33

From the results, we draw the following conclusions.

- $L_2$  distance outperforms cosine distance on the noisy-test evaluation. Both clean validation split and noisy kaggle test proves that.
- Across the evaluated neighborhood sizes,  $k = 30$  yields the strongest performance on the validation stage, suggesting a suitable bias–variance trade-off under the noisy evaluation condition: a too-small  $k$  is sensitive to noisy neighbors, while a too-large  $k$  oversmooths the class boundary. However, on the noisy kaggle testset,  $k = 20$  is the best.
- PCA and LDA can slightly improve validation accuracy, but do not outperform the no-reduction setting on Kaggle. On the clean validation split, both PCA and LDA can provide mild gains by removing redundant dimensions (PCA) or enhancing class separability (LDA).

✓ knn_30_l2pca_256.csv Complete · 18d ago	0.83282	<input type="checkbox"/>
✓ knn_30_l2lda_9.csv Complete · 18d ago	0.52567	<input type="checkbox"/>
✓ knn_30_cospca_256.csv Complete · 18d ago	0.82477	<input type="checkbox"/>
✓ knn_30_l2.csv Complete · 18d ago	0.83786	<input type="checkbox"/>
✓ knn_30_cos.csv Complete · 18d ago · knn 30 cos	0.82980	<input type="checkbox"/>
✓ knn_20_l2pca_256.csv Complete · 1m ago	0.83585	<input type="checkbox"/>
✓ knn_20_l2.csv Complete · 2m ago	0.83987	<input checked="" type="checkbox"/>
✓ knn_20_cos.csv Complete · 3m ago	0.83786	<input type="checkbox"/>

Figure 1: KNN performance on the Kaggle test set.

However, on the noisy test set, dimensionality reduction does not consistently improve performance. In particular, models using LDA show a sharp accuracy drop on Kaggle. A plausible explanation is that LDA is learned in a supervised manner from the clean training distribution, and the learned discriminative subspace may not be invariant to the distribution shift induced by noise.

Once the test features are corrupted, the projected class structure can be significantly distorted, leading to degraded nearest-neighbor decisions. In contrast, using the original feature space without reduction preserves more information and appears more robust under noise.

**NCM.** We further evaluate the performance of the Nearest Class Mean (NCM) classifier on the local validation set. Due to its simplicity, NCM serves as a lightweight baseline for comparison.

We consider two distance metrics, namely the Euclidean distance ( $L_2$ ) and cosine distance. The evaluation results on the validation set are summarized in Table 2.

Table 2: Validation Accuracy of NCM with Different Distance Metrics

Distance Metric	Validation Accuracy (%)
$L_2$ Distance	79.47
Cosine Distance	79.40

From Table 2, we observe that the performance of NCM is relatively stable across different distance metrics, with the Euclidean distance slightly outperforming the cosine distance. However, the overall accuracy of NCM is significantly lower than that of KNN-based methods.

This performance gap can be attributed to the strong modeling assumption of NCM. By representing each class using only a single prototype (the class mean), NCM ignores intra-class variability and higher-order structures in the feature space. As a result, NCM is less capable of handling complex class distributions and noisy features.

Nevertheless, due to its low computational cost and minimal number of parameters, NCM remains attractive in scenarios where efficiency and simplicity are prioritized. In this project, we treat NCM as a baseline method and focus subsequent analysis on more expressive models.

**SVM.** We evaluate Support Vector Machine (SVM) models with different kernel functions and hyper-parameter settings. All SVM models are trained using the One-versus-Rest (OvR) strategy for multi-class classification, and no dimensionality reduction is applied in this section.

Table 3 summarizes the classification accuracy on the clean validation set. For linear SVMs, varying the regularization parameter  $C$  does not lead to significant performance differences, with validation accuracy remaining around 78%–80%. This indicates that a linear decision boundary is insufficient to fully capture the class structure of the feature space.

Table 3: Validation accuracy (%) of SVM models on the clean validation set.

Kernel	Configuration	Accuracy (%)
Linear	$C = 0.1$	79.60
Linear	$C = 0.5$	79.07
Linear	$C = 1.0$	78.27
RBF	$C = 0.1, \gamma = 0.001$	81.67
RBF	$C = 0.5, \gamma = 0.01$	29.87
RBF	$C = 0.5, \gamma = \text{scale}$	84.67
RBF	$C = 0.5, \gamma = \text{auto}$	84.67
Poly	$C = 0.1, d = 2, \gamma = 0.01$	85.60
Poly	$C = 0.5, d = 2, \gamma = 0.01$	85.60
Poly	$C = 0.5, d = 3, \gamma = 0.01$	<b>86.40</b>
Sigmoid	$C = 0.5, \gamma = 0.001$	83.33
Sigmoid	$C = 1.0, \gamma = 0.01$	49.40

In contrast, non-linear kernels exhibit substantially different behaviors. Polynomial kernels achieve the best performance among all tested SVM variants. In particular, the polynomial kernel with degree 3,  $C = 0.5$ , and  $\gamma = 0.01$  reaches a validation accuracy of 86.4%, which is the highest accuracy observed across all SVM experiments. Polynomial kernels are able to model higher-order feature interactions while maintaining relatively stable decision boundaries, which appears well suited for this dataset.

Radial Basis Function (RBF) kernels show highly unstable performance. While the configuration using  $\gamma = \text{scale}$  or  $\gamma = \text{auto}$  achieves reasonable validation accuracy (84.67%), other settings such as  $\gamma = 0.01$  lead to severe performance degradation, with accuracy dropping below 30%. This suggests that RBF kernels are extremely sensitive to the choice of  $\gamma$  and may suffer from overfitting or poor margin behavior under inappropriate parameterization.

Sigmoid kernels demonstrate moderate and inconsistent performance. Although one configuration reaches over 83% accuracy, others collapse to near-random performance, indicating that the sigmoid kernel is less reliable for this task.

We further evaluate selected SVM models on the noisy Kaggle test set. The performance trend differs noticeably from that on the clean validation set. Polynomial SVMs still achieve the best performance among SVM variants, remain the most robust SVM-based models in this experiment.

Submission and Description	Public Score	Select
svm_poly_C0.5_degree2_gamma0.01_coef01.csv Complete · 3m ago	0.81470	<input type="checkbox"/>
svm_poly_C0.5_degree3_gamma0.01_coef01.csv Complete · 3m ago	0.84994	<input checked="" type="checkbox"/>
svm_rbf_C0.5_gammaScale.csv Complete · 23m ago	0.27391	<input type="checkbox"/>
svm_rbf_C0.5_gammaAuto.csv Complete · 23m ago	0.27391	<input type="checkbox"/>
svm_rbf_C0.1_gamma0.001.csv Complete · 1h ago	0.49345	<input type="checkbox"/>

Figure 2: SVM performance on the Kaggle test set.

RBF kernels perform particularly poorly on the noisy test set, with public scores dropping sharply to around 27%–49% depending on the configuration. This confirms that RBF kernels are highly sensitive to noise perturbations, especially when the kernel width parameter  $\gamma$  is not carefully tuned. Linear SVMs exhibit more stable but limited performance, reflecting their restricted expressive power.

**MLP.** We evaluate Multi-Layer Perceptron (MLP) models with different network depths, hidden dimensions, dropout rates, and preprocessing strategies on the clean validation set.

Table 4 summarizes the validation accuracy of the evaluated configurations. Overall, increasing model capacity leads to improved validation performance. Single-hidden-layer models such as MLP-64

already achieve reasonable accuracy, while deeper architectures consistently outperform shallower ones.

Table 4: Validation Accuracy (%) of MLP Models under Different Architectures, Dropout, and Preprocessing (Clean Validation Split).

Model	None	PCA-256	PCA-128
MLP-256	83.80		
MLP-512	84.33		
MLP-512-256	84.47	83.53	82.67
MLP-512-256 (dp=0.1)	83.40	84.13	
MLP-512-256 (dp=0.2)	83.33	83.93	
MLP-512-256-128	<b>85.07</b>	83.73	82.87
MLP-512-256-128 (dp=0.1)	83.33	83.33	
MLP-512-256-128 (dp=0.2)	83.40	82.60	

In particular, the three-layer model MLP-512-256-128 achieves the highest validation accuracy among all tested configurations, indicating that additional representational capacity allows the network to better fit the clean training distribution. Introducing dropout slightly reduces validation accuracy in most cases, suggesting that overfitting on the clean validation set is not severe.

Applying PCA before MLP training generally degrades performance, especially when the feature dimension is aggressively reduced, implying that dimensionality reduction may discard discriminative information useful for classification.

Despite the strong performance on the clean validation set, the best-performing MLP model generalizes poorly to the noisy Kaggle test set, achieving only around 79%–80% accuracy. This significant gap between validation and test performance suggests that the MLP models tend to overfit the clean training distribution and are less robust to feature perturbations introduced by noise, making them less suitable for noisy test scenarios under the current experimental setting.

## 5.2 Overall Conclusion

In this project, we evaluate several statistical learning models under a clean-training / noisy-testing setting. Overall, we find that classical non-parametric methods (e.g., KNN) and kernel-based methods (SVM) are more robust to the distribution shift introduced by test-time noise than neural models trained on clean data only. Among all submitted models, **the polynomial-kernel SVM achieves the best Kaggle performance, reaching an accuracy close to 85%**, which indicates that a moderately expressive non-linear decision boundary can effectively capture class structure while remaining relatively stable under feature perturbations. In comparison, although MLP attains strong accuracy on the clean validation set, its Kaggle accuracy drops to around 79%–80%, suggesting overfitting to the clean training distribution and limited robustness to noise. These results highlight the importance of robustness-aware model selection for noisy real-world evaluation, where the best clean-validation model may not transfer to the noisy test distribution.