18006555_CW2_SL_COMP0078_code

January 3, 2024

COMP0078 Supervised Learning - Coursework 1

Student ID: 18006555

Part I [20%]

Rademacher Complexity of finite Spaces

We start with $X_1, \cdots, X_m$ of centered random variables ($E[X_i] = 0$), taking values in $[a, b]$

1.1

We let $\bar{X} = \max_i X_i$. For $\lambda > 0$, we aim to show:

$$E[\bar{X}] \leq \frac{1}{\lambda} log E[e^{\lambda \bar{X}}]$$

We use Jensen's inequality i.e. if g is a convex function and X is a random variable, then $g(E[X]) \leq E[g(X)]$.

Since $g(x) = e^{\lambda x}$ is a convex function for $\lambda > 0$, we note that:

$$e^{\lambda E[\bar{X}]} \leq E[e^{\lambda \bar{X}}]$$
$$\lambda E[\bar{X}] \leq \log E[e^{\lambda \bar{X}}]$$
$$E[\bar{X}] \leq \frac{1}{\lambda} \log E[e^{\lambda \bar{X}}]$$

1.2

We now aim to show:

$$\frac{1}{\lambda} \log E[e^{\lambda \bar{X}}] \leq \frac{1}{\lambda} \log m + \lambda \frac{(b-a)^2}{8}$$

We use Hoeffding's Lemma which states that for any random variable X such that $X - E[X] \in [a, b]$ and $\lambda > 0$, we have:

$$E[e^{\lambda (X - E[X])}] \leq e^{\frac{1}{8}\lambda^2 (b-a)^2}$$

If we choose X_i as our random variable, this satisfies our conditions for Hoeffding's Lemma since $X_i - E[X_i] \in [a, b]$ since $E[X_i] = 0$. We then apply the Lemma over a summation of the X_i s from 1 to m:

$$E[e^{\lambda(X_i - E[X_i])}] \leq e^{\frac{1}{8}\lambda^2(b-a)^2}$$
$$\sum_{i=1}^{m} E[e^{\lambda(X_i)}] \leq \sum_{i=1}^{m} e^{\frac{1}{8}\lambda^2(b-a)^2}$$
$$E[e^{\lambda\bar{X}}] \leq \sum_{i=1}^{m} E[e^{\lambda(X_i)}] \leq m \times e^{\frac{1}{8}\lambda^2(b-a)^2}$$
$$\frac{1}{\lambda}\log E[e^{\lambda\bar{X}}] \leq \frac{1}{\lambda}\log m + \frac{1}{\lambda}\log e^{\frac{1}{8}\lambda^2(b-a)^2}$$
$$\frac{1}{\lambda}\log E[e^{\lambda\bar{X}}] \leq \frac{1}{\lambda}\log m + \frac{1}{8}\lambda(b-a)^2$$

1.3

Now we combine the previous two inequality results, and choose $\lambda = \sqrt{\frac{8\log m}{(b-a)^2}}$, to conclude with:

$$E[\bar{X}] \leq \frac{1}{\lambda}\log m + \frac{1}{8}\lambda(b-a)^2$$
$$E[\bar{X}] \leq \sqrt{\frac{(b-a)^2}{8\log m}}\log m + \frac{1}{8}\sqrt{\frac{8\log m}{(b-a)^2}}(b-a)^2$$
$$E[\bar{X}] \leq \sqrt{\frac{1}{8}(b-a)^2\log m} + \sqrt{\frac{1}{8}(b-a)^2\log m}$$
$$E[\bar{X}] \leq 2\sqrt{\frac{1}{8}(b-a)^2\log m}$$
$$E[\max_{i=1,\cdots,m} X_i] \leq \frac{b-a}{2}\sqrt{2\log m}$$

1.4

We now provide the bound for the Rademacher complexity of a finite set of hypotheses. For each point $x \in S$, $\frac{1}{n}\sum_{j=1}^{n}\sigma_j x_j$ is a sum of independent random variables, which are centered because Rademacher variables σ_j have zero mean, as they are equally likely to be -1 or 1. Therefore each term $\sigma_j x_j$ is either $-|x_j|$ or $|x_j|$, and hence the range is $2|x_j|$.

We have already showed that for centered random variables X_i which are bounded by $[a, b]$, that $E[\max_i X_i] \leq \frac{b-a}{2}\sqrt{2\log m}$, where $b - a = 2|x_j|$ for each term in the sum.

Now we continue as:

$$R(S) = E_\sigma \max_{x\in S}\frac{1}{n}\sum_{j=1}^{n}\sigma_j x_j \leq \max_{x\in S}\frac{1}{n}\sum_{j=1}^{n}|x_j|\sqrt{2\log m} = \max_{x\in S}||x||_2\frac{\sqrt{2\log m}}{n}$$

$$R(S) < \max_{x\in S}||x||_2\frac{\sqrt{2\log m}}{n}$$

1.5

We start with the definition for the empirical Rademacher complexity $R_S(H)$, where H is a set of hypotheses, and has finite cardinality $|H| < +\infty$. We can use the previous sections to prove an upper bound for $R_S(H)$, where $|H|$ appears logarithmically:

$$R_S(H) = E_\sigma \left[sup_{f \in H} \frac{1}{n} \sum_{i=1}^{n} \sigma_i \cdot f(x_i) \right]$$

Now we consider each hypothesis $f \in H$ as a vector in R^n, defined by its evaluations on the set of points $S = (x_i)_{i=1}^n$ i.e. we write $f_S = (f(x_i))_{i=1}^n$. f_S is also just a set of points like S, hence we can apply the result from 1.4 on f_S:

$$R(f_S) \leq \max_{f \in H} ||f_S||_2 \frac{\sqrt{2 \log m}}{n}$$

The empirical Rademacher complexity for the hypothesis set H is the expectation of the supremum over all such bounds for individual hypotheses f i.e. we are looking for the 'worst-case' for the bound across all hypotheses in H. Hence:

$$R_S(H) \leq \frac{M}{n} \sqrt{2 \log m}$$

Where $M = \max_{f \in H} ||f_S||_2$ i.e. $\forall f \in H, ||f_S||_2 \leq M$. Since we are now dealing with a set of hypotheses H, and the empirical Rademache complexity $R_S(H)$ now represents a measure of complexity of H, and depends on how many different hypotheses f wer are considering i.e. $|H|$, as opposed to 1.4 where we were considering the complexity of the set of points S, determined by $|S| = m$. Hence we finish with the upper bound for $R_S(H)$ as follows:

$$R_S(H) \leq \frac{M}{n} \sqrt{2 \log |H|}$$

Part II [40%]

Bayes Decision Rule and Surrogate Approaches

2.1

We start with the misclassification error of a classification rule $c(x)$, defined as follows:

$$R(c) = P_{(x,y) \sim \rho}(c(x) \neq y)$$

Since the probability of an event occurring is equal to the expected value of its corresponding indicator function, we can write $R(c)$ as follows:

$$R(c) = E_{(x,y) \sim \rho}[1_{c(x) \neq y}]$$

Now using the definition of Expectation, we can expand as follows:

$$R(c) = \int_{X \times Y} 1_{c(x) \neq y} d\rho(x, y)$$

This integral effectively sums the occurrences where $c(x) \neq y$ i.e. the misclassifications, which in turn will correspond to the average number of misclassifications (as a proportion of the total input-output pairs (x, y)) and hence represents the misclassification error $R(c)$.

2.2

We now solve Surrogate Approaches for different loss functions, calculating the closed-form of the minimiser f_* of $\epsilon(f)$ defined as:

$$\epsilon(f) = \int_{X \times Y} l(f(x), y) d\rho(x, y) = \int_X \left(\int_Y l(f(x), y) d\rho(y|x) \right) d\rho_X(x)$$

We now consider the problem in the inner integral point-wise $\forall x \in X$, differentiating with respect to f and setting equal to zero:

$$\frac{\partial}{\partial f} \int_Y l(f(x), y) d\rho(y|x) = 0$$
$$\int_Y \frac{\partial l(f(x), y)}{\partial f} d\rho(y|x) = 0$$

2.2 part a) Squared Loss: $l(f(x), y) = (f(x) - y)^2$

$$\begin{aligned}
0 &= \int_Y \frac{\partial l(f(x), y)}{\partial f} d\rho(y|x) \\
0 &= \int_Y \frac{\partial (f(x) - y)^2)}{\partial f} d\rho(y|x) \\
0 &= \int_Y 2(f(x) - y) d\rho(y|x) \\
f(x) &= \int_Y y d\rho(y|x) \\
f(x) &= (1)\rho(y = 1|x) + (-1)\rho(y = -1|x) \\
f_*(x) &= 2\rho(y = 1|x) - 1 \quad (= E[y|x])
\end{aligned}$$

2.2 part b)

Exponential loss: $l(f(x), y) = \exp(-yf(x))$

$$
\begin{aligned}
0 &= \int_Y \frac{\partial l(f(x), y)}{\partial f} d\rho(y|x) \\
0 &= \int_Y \frac{\partial \exp(-yf(x))}{\partial f} d\rho(y|x) \\
0 &= \int_Y -y \exp(-yf(x)) d\rho(y|x) \\
0 &= -\exp(-f(x))\rho(y=1|x) + \exp(f(x))\rho(y=-1|x) \\
\exp(-f(x))\rho(y=1|x) &= \exp(f(x))\rho(y=-1|x) \\
\exp(f_*(x))^2 &= \frac{\rho(y=1|x)}{\rho(y=-1|x)} \\
f_*(x) &= \frac{1}{2} \log\left(\frac{\rho(y=1|x)}{1-\rho(y=1|x)}\right)
\end{aligned}
$$

2.2 part c)

Logistic loss: $l(f(x), y) = \log(1 + \exp(-yf(x)))$

$$
\begin{aligned}
0 &= \int_Y \frac{\partial l(f(x), y)}{\partial f} d\rho(y|x) \\
0 &= \int_Y \frac{\partial \log(1 + \exp(-yf(x)))}{\partial f} d\rho(y|x) \\
0 &= \int_Y \frac{-y}{1 + \exp(yf(x))} d\rho(y|x) \\
0 &= \frac{-1}{1 + \exp(f(x))}\rho(y=1|x) + \frac{1}{1 + \exp(-f(x))}\rho(y=-1|x) \\
(1 + \exp(-f(x)))\rho(y=1|x) &= (1 + \exp(f(x))\rho(y=-1|x)) \\
\frac{1 + \exp(f(x))}{\exp(-f(x)) + 1} &= \frac{\rho(y=1|x)}{1-\rho(y=1|x)} \\
\exp(f(x)) &= \frac{\rho(y=1|x)}{1-\rho(y=1|x)} \\
f_*(x) &= \log\left(\frac{\rho(y=1|x)}{1-\rho(y=1|x)}\right)
\end{aligned}
$$

2.2 part d)

Hinge Loss $l(f(x), y) = max(0, 1 - yf(x))$

$$
\begin{aligned}
0 &= \int_Y \frac{\partial l(f(x), y)}{\partial f} d\rho(y|x) \\
0 &= \int_Y \frac{\partial \max(0, 1 - yf(x))}{\partial f} d\rho(y|x) \\
0 &= \frac{\partial \max(0, 1 - f(x))}{\partial f} \rho(y = 1|x) + \frac{\partial \max(0, 1 + f(x))}{\partial f} \rho(y = -1|x) \\
0 &= \int_Y -y \cdot 1_{(yf(x)<1)} d\rho(y|x) \\
0 &= -1_{(f(x)<1)} \rho(y = 1|x) + 1_{(f(x)>-1)} \rho(y = -1|x) \\
1_{(f_*(x)<1)} \rho(y = 1|x) &= 1_{(f_*(x)>-1)} (1 - \rho(y = 1|x))
\end{aligned}
$$

This can also be expressed as an expectation:

$$E\left[-y \cdot 1_{(yf(x)<1)}|x\right]$$

We choose $f_*(x)$ such that it "leans" towards 1 or -1 depending on which of $\rho(y = 1|x)$ or $1-\rho(y = 1|x)$ is larger, to minimise the respective side of the above equation. However we also notice that if $f_*(x) > 1$, then the equation simplifies to indicate $\rho(y = 1|x) = 1$, and similarly if $f_*(x) < -1$ then we observe $\rho(y = 1|x) = 0$. When $-1 < f_*(x) < 1$, then $\rho(y = 1|x) = \rho(y = -1|x) = \frac{1}{2}$, hence informing us of constraints on ρ.

2.3

The Bayes decision rule c_* which minimises $R(c)$ over all possible decision rules $c : X \to \{-1, 1\}$ is defined as follows:

$$c^*(x) = \begin{cases} 1 & \text{if } \rho(y = 1|x) \geq \rho(y = -1|x) \\ -1 & \text{otherwise} \end{cases}$$

Where the decision rule maps $x = 0$ to 1 in the special case where $\rho(y = 1|x) = \rho(y = -1|x)$, for simplicity.

2.4

Now we aim to understand if the surrogate frameworks in 2.2 are Fisher consistent i.e. we can find a map $d : R \to \{-1, 1\}$ such that $R(c_*(x)) = R(d(f_*(x)))$.

For the Squared Loss $l(f(x), y) = (f(x) - y)^2$, this is not Fisher consistent as the obtained $f_*(x) = 2\rho(y = 1|x) - 1$ captures the mean instead of the mode, and hence may not correctly reflect the underlying class probabilities within the binary classification.

For the Exponential loss $l(f(x), y) = \exp(-yf(x))$ and Logistic loss $l(f(x), y) = \log(1 + \exp(-yf(x)))$ functions, we choose $d(f) = sign(f)$ also, since $f_*(x)$ for both surrogate frameworks are such that $R(c_*(x)) = R(d(f_*(x)))$. We define the $sign(x)$ function as follows:

$$sign(f) = \begin{cases} 1 & f \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

For the Exponential loss, this works as it penalises misclassification exponentially, and hence leads to $d(f(x)) = sign(f(x))$ being effective because the loss function tends to create a strong separation between classes. Similar reasoning applies for the Logistic loss, as the logistic function transforms $f_*(x)$ into a probability.

For the Hinge loss $l(f(x), y) = max(0, 1 - yf(x))$, we also choose $d(f) = sign(f)$, as the loss function focuses on penalising points on the wrong side of the margin or even too close to the decision boundary. Again, there tends to be a strong separation between classes.

2.5.1

We prove the first intermediate step as follows:

$$\begin{aligned} |R(sign(f)) - R(sign(f_*))| &= |E[1_{\{sign(f(x))\neq y\}}] - E[1_{\{sign(f_*(x))\neq y\}}]| \\ &= \int_{X\times Y} 1_{\{sign(f(x))\neq y\}} - 1_{\{sign(f_*(x))\neq y\}} d\rho(x, y) \end{aligned}$$

With $Y = \{-1, 1\}$, we know that the integrand will be non-zero when $sign(f(x)) \neq sign(f_*(x))$ i.e. only on X_f $= \{x \in X | sign(f(x)) \neq sign(f_*(x))\}$. Hence:

$$|R(sign(f)) - R(sign(f_*))| = \int_{X_f} 1_{\{sign(f(x))\neq y\}} - 1_{\{sign(f_*(x))\neq y\}} d\rho_X(x)$$

Over X_f, the risk difference is affected by how often f and f_* misclassify, and the magnitude of their differences. $|f_*(x)|$ indicates the 'confidence' of the prediction by f_*, since it quantifies how far the prediction is from the decision boundary (which is at 0 for the sign function). Hence we see the integral reduce to the desired result:

$$|R(sign(f)) - R(sign(f_*))| = \int_{X_f} |f_*(x)| d\rho_X(x)$$

2.5.2

We continue the intermediate steps as follows. Looking at the first inequality, $|f_*(x) - f(x)|$ represents the distance between $f_*(x)$ and $f(x)$ at each point x. Since $|f_*(x)|$ is the distance from $f_*(x)$ to the decision boundary (zero), and when on X_f (where $sign(f(x)) \neq sign(f_*(x))$), we know that $f(x)$ and $f_*(x)$ are on opposite sides of the decision boundary, and hence $|f_*(x) - f(x)|$ is at least as large as the distance of either point to the decision boundary. Hence $|f_*(x)| \leq |f_*(x) - f(x)|$, and we continue as follows:

$$\int_{X_f} |f_*(x)| d\rho_X(x) \leq \int_{X_f} |f_*(x) - f(x)| d\rho_X(x)$$

For the second inequality, we apply Jensen's inequality where for a convex function g, we have $E[g(X)] \geq g(E[X])$. By letting $g(X) = X^2$ which is convex, and taking the square root on both sides, we obtain the result $\sqrt{E[X^2]} \geq E[X]$ for a non-negative random variable X.

Let $X = |f_*(x) - f(x)|$, and we have the following:

$$E[|f_*(x) - f(x)|] \leq \sqrt{E[|f_*(x) - f(x)|^2]}$$

We also note that by simple definition of expectation, that $\int_{X_f} |f_*(x) - f(x)| d\rho_X(x) = E[|f_*(x) - f(x)|]$. Combining all results, we have the following:

$$\int_{X_f} |f_*(x)| d\rho_X(x) \leq \int_{X_f} |f_*(x) - f(x)| d\rho_X(x) \leq \sqrt{E[|f_*(x) - f(x)|^2]}$$

2.5.3

Finally, we start with the following:

$$\epsilon(f) - \epsilon(f_*) = \int_{X \times Y} l(f(x), y) - l(f_*(x), y) d\rho(x, y)$$

Once again, we notice that the losses will cancel each other out when $f(x) = f_*(x)$, hence we are left with the cases on X_f. Since we are using the squared loss $l(f(x), y) = (f(x) - y)^2$, we can simplify as follows:

$$\begin{aligned}
\epsilon(f) - \epsilon(f_*) &= \int_{X_f} (f(x) - y)^2 - (f_*(x) - y)^2 \; d\rho_X(x) \\
&= \int_{X_f} f(x)^2 - 2yf(x) + y^2 - f_*(x)^2 + 2yf_*(x) - y^2 \; d\rho_X(x) \\
&= \int_{X_f} f(x)^2 - f_*(x)^2 - 2y(f(x) - f_*(x)) d\rho_X(x) \\
&= \int_{X_f} f(x)^2 - 2f(x)f_*(x) + f_*(x)^2 - 2y(f(x) - f_*(x)) + 2f(x)f_*(x) - 2f_*(x)^2 d\rho_X(x) \\
&= \int_{X_f} (f(x) - f_*(x))^2 - 2y(f(x) - f_*(x)) + 2f_*(x)(f(x) - f_*(x)) d\rho_X(x) \\
&= \int_{X_f} (f(x) - f_*(x))^2 d\rho_X(x) + 2 \int_{X_f} (f_*(x) - 2y)(f(x) - f_*(x)) d\rho_X(x) \\
&= E[|f(x) - f_*(x)|^2] + 2E[(f_*(x) - 2y)(f(x) - f_*(x))]
\end{aligned}$$

Since f_* is the minimiser of $\epsilon(f)$, therefore any variations around f_* i.e. the quantity $(f(x) - f_*(x))$ integrated over the distribution ρ should all cancel out, resulting in just the first Expectation term. Hence:

$$\epsilon(f) - \epsilon(f_*) = E[|f(x) - f_*(x)|^2]$$

Combining all results across 2.5, we obtain the final result: (the first inequality to show ≥ 0 is trivial as f_* is the minimiser of $\epsilon(f)$)

$$0 \leq R(sign(f)) - R(sign(f_*)) \leq \sqrt{\epsilon(f) - \epsilon(f_*)}$$

Part III [40%]

0.0.1 Exploration of Kernel Perceptron (Handwritten Digit Classification)

In this report, we aim to apply supervised learning techniques to create a classifier for handwritten digits from 0 to 9. We aim to use the perceptron, generalising the perceptron to use kernel functions, and also to employ a One-vs-Rest (OvR) approach to seperate between 2 classes, before then separating k classes.

We aim to use a polynomial and gaussian kernel function, with parameters degree d and kernel width c respectively, defined as follows:

$$K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p}, \mathbf{q})^d$$
$$K_c(\mathbf{p}, \mathbf{q}) = \exp\left(-c||\mathbf{p} - \mathbf{q}||^2\right)$$

[ ]:
```
# Importing Python modules
import numpy as np
import pandas as pd
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from IPython.display import display, Markdown, Math, Latex
import warnings
from abc import abstractmethod, ABC
warnings.simplefilter(action='ignore', category=FutureWarning)
%matplotlib inline
```

[ ]:
```
df = pd.read_csv('zipcombo.dat', delim_whitespace=True, header=None)
```

[ ]:
```
class KernelPerceptron(ABC):

    @abstractmethod
    def fit(self, X_train: np.array, y_train: np.array, d: int):
        pass

    @abstractmethod
    def evaluate(self, X_train: np.array, X_test: np.array, y_test: np.array, d:
 ↪ int):
        pass
```

[ ]:
```
class KPOVr(KernelPerceptron):

    def __init__(self, num_classifiers: int, max_epochs: int, shuffle: bool,␣
 ↪epsilon: float):
        self.num_classifiers = num_classifiers
        self.max_epochs = max(max_epochs, 10)
        self.shuffle = shuffle
        self.epsilon = epsilon
        self.alpha = None
        self.kernel_func = None
```

```
  def fit(self, X_train, y_train):
      gram_matrix = self.kernel_func(X_train.values, X_train.values)
      self.alpha = np.zeros((self.num_classifiers, X_train.shape[0]))
      train_errors = []
      for i in range(self.max_epochs):
          train_mistakes = 0
          if self.shuffle:
              indices = np.random.choice(X_train.shape[0], X_train.shape[0],␣
↪replace=False)
          else:
              indices = np.arange(0, X_train.shape[0], 1)
          for example in indices:
              y_hat_list = self.alpha @ gram_matrix[:, example]
              y_hat_label = np.argmax(y_hat_list)
              # Raw label comparison
              if y_train.values[example] != y_hat_label:
                  self.alpha[y_train.values[example], example] += 1
                  self.alpha[y_hat_label, example] -= 1
                  train_mistakes += 1
          train_error = train_mistakes / X_train.shape[0] * 100
          train_errors.append(train_error)
          if i > 10:
              if np.abs(np.mean(train_errors[i-5:i])-np.mean(train_errors[i-4:
↪i])) < self.epsilon:
                  return train_errors[-1]
      return train_errors[-1]

  def evaluate(self, X_train, X_test, y_test, confusion_matrix_print: bool,␣
↪misclassified_points: bool):
      gram_matrix = self.kernel_func(X_train.values, X_test.values)
      y_hat_label = np.argmax(self.alpha @ gram_matrix, axis=0)
      test_error = sum(y_hat_label != y_test.values) / X_test.shape[0] * 100
      incorrectly_classified_indices = []  # To store the original indices of␣
↪misclassified test data
      if misclassified_points:
          y_test_values = y_test.values.flatten()  # Flatten the y_test␣
↪DataFrame
          incorrectly_classified_indices = y_test.index[y_test_values !=␣
↪y_hat_label].tolist()
      if confusion_matrix_print:
          confusion_matrix = np.zeros((self.num_classifiers, self.
↪num_classifiers))
          for i, j in zip(y_test.values, y_hat_label):
              if i != j:
                  # Mistake
```

```
                    confusion_matrix[i, j] += 1
            confusion_matrix = np.nan_to_num(confusion_matrix /␣
↪confusion_matrix.sum(axis=1)[:, np.newaxis])

        if misclassified_points and confusion_matrix_print:
            return test_error, confusion_matrix, incorrectly_classified_indices
        elif misclassified_points:
            return test_error, incorrectly_classified_indices
        elif confusion_matrix_print:
            return test_error, confusion_matrix
        else:
            return test_error

    def poly_gram_matrix(self, d):
        def kernel_func(x, y):
            return (x @ y.T) ** d

        self.kernel_func = kernel_func

    def gaussian_gram_matrix(self, c):
        def kernel_func(x, y):
            dist_matrix = cdist(x, y)
            return np.exp(-c * dist_matrix**2)

        self.kernel_func = kernel_func
```

[ ]:
```
def train_test_split(df, shuffle_df=True, split_ratio=0.8):
    split_idx = int(len(df) * split_ratio)
    if shuffle_df:
        df_shf = df.sample(frac=1.0)
    train_idx, test_idx = df_shf[:split_idx], df_shf[split_idx:]

    # Correct column indexing to start from 0 for Y (outputs)
    X_train, y_train = train_idx.iloc[:, 1:], train_idx.iloc[:, 0].astype(int)
    X_test, y_test = test_idx.iloc[:, 1:], test_idx.iloc[:, 0].astype(int)
    return X_train, y_train, X_test, y_test
```

0.0.2 3.1: Basic Results

First, we used the polynomial kernel function without cross validation to compute training and testing errors for degrees d from $\{1, 2, \cdots, 7\}$, displaying the mean μ and standard deviation over 20 runs.

d (degree)	**Train Error** $(\mu \pm \sigma)$ in %	**Test Error** $(\mu \pm \sigma)$ in %
1	5.7045 ± 0.2753	8.6237 ± 0.9705
2	0.2978 ± 0.0825	3.5618 ± 0.4188
3	0.0867 ± 0.0447	3.2903 ± 0.3605

d (degree)	**Train Error** $(\mu \pm \sigma)$ in %	**Test Error** $(\mu \pm \sigma)$ in %
4	0.0592 ± 0.0453	3.1667 ± 0.3318
5	0.0524 ± 0.0427	3.1505 ± 0.5811
6	0.0612 ± 0.0409	2.9462 ± 0.4286
7	0.0504 ± 0.0505	3.0780 ± 0.3939

Here, we observe that the degrees d which seem to obtain the lowest test errors are in the region $4 \leq d \leq 7$. $d = 1$ stands out with a significantly large test error of 8%+. This suggests that a polynomial degree of 1 is significantly unable to capture the patterns and intricacies of digits, which appears sensical as the images are in 2D. For $d = 2$ onwards, the errors are similar yet still decreasing for larger d, for both training and testing. With an error rate of ~3%, this suggests a good start to a strong model, and motivates cross validation to identify the optimal degree d. We must also be wary of overfitting for higher degrees d, hence we may lean towards $d = 4$ or $d = 5$ as there is lesser improvement to the test error for $d > 5$.

[ ]:
```
def question3_1(model, degrees, df, confusion_matrix_print: bool,␣
↪misclassified_points: bool, num_runs=20):
    results = {d: {'train_errors': [], 'test_errors': []} for d in degrees}
    for d in degrees:
        model.poly_gram_matrix(d)
        for run in range(num_runs):
            X_train, y_train, X_test, y_test = train_test_split(df)
            # Train perceptrons for the current degree (d) and calculate error␣
↪rates
            train_error = model.fit(X_train, y_train)
            test_error = model.evaluate(X_train, X_test, y_test,␣
↪confusion_matrix, misclassified_points)
            # print('train_error =', train_error, '; test_error =', test_error)

            results[d]['train_errors'].append(train_error)
            results[d]['test_errors'].append(test_error)
    return results
```

[ ]:
```
model = KPOVr(10, 15, True, 0.001)
degrees = range(1,8)
results3_1 = question3_1(model, degrees, df, False, False, num_runs=20)

for d in degrees:
    train_mean = np.mean(results3_1[d]['train_errors'])
    train_std = np.std(results3_1[d]['train_errors'])
    test_mean = np.mean(results3_1[d]['test_errors'])
    test_std = np.std(results3_1[d]['test_errors'])

    print(f"d = {d}: Train Error = {train_mean:.4f} ± {train_std:.4f}, Test␣
↪Error = {test_mean:.4f} ± {test_std:.4f}")
```

0.0.3 3.2 & 3.3: Cross-validation & Confusion matrix

We now perform 5-fold cross-validation to select the best degree parameter d^*. We then retrain the model using d^* to identify the test errors on the full data set again, over 20 runs.

Mean d^* with std	**Test Error** $(\mu \pm \sigma)$ in %
4.7 ± 1.1000	3.1452 ± 0.3246

Here we identify the mean d^* similarly to our ideas from 3.1. However the standard deviation is relatively high, also confirming that degrees in the range $4 \leq d \leq 7$ are producing similar results.

We now provide a confusion matrix, providing the errors for digit a being mistaken for digit b, averaged over 20 runs in which the optimal d^* was found via cross-validation and used.

	0	1	2	3	4	5	6	7	8	9
0	0.000 ± 0.000	0.020 ± 0.060	0.154 ± 0.170	0.174 ± 0.152	0.064 ± 0.109	0.130 ± 0.164	0.222 ± 0.239	0.037 ± 0.074	0.100 ± 0.300	0.100 ± 0.170
1	0.000 ± 0.000	0.000 ± 0.000	0.100 ± 0.200	0.000 ± 0.000	0.250 ± 0.261	0.000 ± 0.000	0.250 ± 0.344	0.100 ± 0.300	0.100 ± 0.200	0.000 ± 0.000
2	0.083 ± 0.117	0.088 ± 0.116	0.000 ± 0.000	0.093 ± 0.120	0.237 ± 0.174	0.010 ± 0.030	0.074 ± 0.180	0.207 ± 0.129	0.180 ± 0.170	0.029 ± 0.057
3	0.031 ± 0.065	0.018 ± 0.037	0.121 ± 0.119	0.000 ± 0.000	0.011 ± 0.033	0.502 ± 0.169	0.000 ± 0.000	0.080 ± 0.092	0.219 ± 0.178	0.018 ± 0.055
4	0.021 ± 0.046	0.159 ± 0.168	0.235 ± 0.119	0.055 ± 0.081	0.000 ± 0.000	0.090 ± 0.126	0.153 ± 0.134	0.053 ± 0.083	0.053 ± 0.057	0.180 ± 0.122
5	0.221 ± 0.180	0.000 ± 0.000	0.036 ± 0.063	0.230 ± 0.132	0.054 ± 0.107	0.000 ± 0.000	0.270 ± 0.192	0.019 ± 0.040	0.100 ± 0.095	0.070 ± 0.120
6	0.380 ± 0.293	0.087 ± 0.129	0.140 ± 0.166	0.000 ± 0.000	0.257 ± 0.207	0.127 ± 0.144	0.000 ± 0.000	0.000 ± 0.027	0.009 ± 0.000	0.000 ± 0.000
7	0.000 ± 0.000	0.053 ± 0.111	0.189 ± 0.206	0.069 ± 0.117	0.240 ± 0.201	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.139 ± 0.198	0.308 ± 0.247
8	0.116 ± 0.148	0.066 ± 0.086	0.171 ± 0.164	0.267 ± 0.125	0.144 ± 0.158	0.078 ± 0.073	0.037 ± 0.057	0.051 ± 0.084	0.000 ± 0.000	0.070 ± 0.075
9	0.020 ± 0.060	0.000 ± 0.000	0.110 ± 0.097	0.070 ± 0.155	0.405 ± 0.237	0.037 ± 0.075	0.000 ± 0.000	0.359 ± 0.276	0.000 ± 0.000	0.000 ± 0.000

We observe that there are indeed zeros across the diagonal, as correctly identified digits will have

zero error by definition. There doesn't appear to be any significant trends or particular differences between digits, however upon adding the columns of each row which represent how many times that digit was misclassified, we identify that the digit 4 has a particular high error rate and the digit 1 has a particular low error rate. Of course, we cannot strictly add each number as these are rates, but we still gain a sense of which digits are more difficult to analyse than others. With 4 being the most erroneous, due to its complexity and from even having multiple ways to write it, and 1 being the les erroneous due to its simplicity, the results seem to match intuition here.

[ ]:
```
def question3_2(model, degrees, df, confusion_matrix_print: bool,␣
↪misclassified_points: bool, num_runs=20, num_folds=5):
    best_ds = []  # To store the best d for each run
    test_errors = []  # To store the test errors for each run
    confusion_matrices = []  # To store the confusion matrices for each run

    for run in range(num_runs):
        print(f"run = {run+1}")
        X_train, y_train, X_test, y_test = train_test_split(df)
        fold_size = len(X_train) // num_folds
        best_d = None
        best_validation_error = float('inf')

        for d in degrees:
            print(f"d = {d}")
            model.poly_gram_matrix(d)
            validation_errors = []

            for fold in range(num_folds):
                start = fold * fold_size
                end = (fold + 1) * fold_size
                X_fold = X_train.iloc[start:end]
                y_fold = y_train.iloc[start:end]
                mask = ~X_train.index.isin(range(start, end))
                X_remainder = X_train[mask]
                y_remainder = y_train[mask]

                model.fit(X_remainder, y_remainder)
                validation_error = model.evaluate(X_remainder, X_fold, y_fold,␣
↪False, False)
                validation_errors.append(validation_error)

            mean_validation_error = np.mean(validation_errors)
            if mean_validation_error < best_validation_error:
                best_validation_error = mean_validation_error
                best_d = d

        # Retrain with the best d on the full 80% training set
        model.poly_gram_matrix(best_d)
```

```
        model.fit(X_train, y_train)
        test_error, confusion_matrix = model.evaluate(X_train, X_test, y_test,␣
 ↪confusion_matrix_print, misclassified_points)
        print(f"Best d: {best_d}, Test Error: {test_error:.4f}\n")
        best_ds.append(best_d)
        test_errors.append(test_error)
        confusion_matrices.append(confusion_matrix)

    return best_ds, test_errors, confusion_matrices
```

[ ]:
```
# Perform cross-validation
model = KPOVr(10, 15, True, 0.001)
degrees = range(1, 8)
best_ds, test_errors, confusion_matrices = question3_2(model, degrees, df,␣
 ↪True, False, num_runs=10, num_folds=5)
confusion_matrices = np.array(confusion_matrices)

mean_best_d = np.mean(best_ds)
std_best_d = np.std(best_ds)
mean_test_error = np.mean(test_errors)
std_test_error = np.std(test_errors)
mean_confusion_matrix = np.mean(confusion_matrices, axis=0)
std_confusion_matrix = np.std(confusion_matrices, axis=0)

print(f"Mean Best d: {mean_best_d:.4f} ± {std_best_d:.4f}")
print(f"Mean Test Error: {mean_test_error:.4f} ± {std_test_error:.4f}")

print("Mean Confusion Matrix:")
display(pd.DataFrame(mean_confusion_matrix))
print("\nStandard Deviation Confusion Matrix:")
display(pd.DataFrame(std_confusion_matrix))
```

0.0.4 3.4: Hardest images to predict

We aim to identify the top 5 hardest images to predict correctly from the entire dataset. We achieve this by training the model over 100 runs, and counting the frequency of each image being predicted incorrectly in the testing phase, and divide by the number of times that each image appeared in the test set at all. We increased the number of runs from 20 to 100, as random shuffles of the dataset mean that some images may be tested more often than others, hence increasing the number of runs allows a fairer test to occur. We trained the model with $d = 5$, assuming it as the best degree from the prior cross-validation.

We first sort by the misclassification ratios which are the highest, and then by the number of times it was misclassified, if the ratios were the same. On testing, it showed ~50 images had a 100% misclassification rate, hence the need to narrow down further. We could reduce this pool by increasing the number of runs much above 100, however we still understand that the images of these digits are not particularly 'clear' or 'well-drawn' and I find it unsurprising that these are hard to predict. With digits completely stretched and warped, it would require a model which could

identify much more finer details and patterns. This is not achievable by a perceptron algorithm, perhaps a convolutional neural network would be more success.

Zooming in further, the top 5 hardest to predict digits included an vertically stretched 6 and 4, which the algorithm may understandly perceive as a 1. The model would need to distinguish between regions with only 3 pixels, to say, understand if there is a 'hole' in the 6 or 4 or not, which is possible for the human eye but without a cNN or potentially higher degree d is unfeasible. In addition, a correct label of zero was attributed to what even a human would perceive as a horizontal smudged line. There is another example (which did not appear in my top 5) which is an almost indistinguishable example to this, but has 7 as the correct label. This reassures the model's strength as it is almost better to predict these (albeit minority) images as incorrect, to not overfit and learn the wrong 'habits'.

[ ]:
```
def question3_4(model, d, df, confusion_matrix_print: bool,␣
↪misclassified_points: bool, num_runs=20):
    num_points = len(df)
    misclassified_counts = np.zeros(num_points)
    appearance_counts = np.zeros(num_points)
    model.poly_gram_matrix(d)
    for run in range(num_runs):
        X_train, y_train, X_test, y_test = train_test_split(df)
        model.fit(X_train, y_train)
        # Calculate the error rate for the testing phase and count␣
↪misclassified points
        _, incorrectly_classified_indices = model.evaluate(X_train, X_test,␣
↪y_test, False, True)
        misclassified_counts[incorrectly_classified_indices] += 1
        appearance_counts[X_test.index] += 1
    # Avoid division by zero
    appearance_counts[appearance_counts == 0] = 1
    return misclassified_counts, appearance_counts
```

[ ]:
```
model = KPOVr(10, 15, True, 0.001)
degree = 5
misclassified_counts, appearance_counts = question3_4(model, degree, df, False,␣
↪True, num_runs=100)
```

[ ]:
```
misclassification_ratios = misclassified_counts / appearance_counts

# Custom sorting key
sorting_key = np.lexsort((-misclassification_ratios, -misclassified_counts))

top_N = 5
top_5_indices = sorting_key[:top_N]

# Print the results
for index in top_5_indices:
```

```
    print(f"Index {index}: Misclassification Ratio␣
 ↪{misclassification_ratios[index]}, Misclassified Count␣
 ↪{misclassified_counts[index]}")

top5_hardest_images = df.iloc[:, 1:].iloc[top_5_indices]
top5_hardest_labels = df.iloc[:, 0].iloc[top_5_indices]

# Visualize the top 5 hardest-to-predict images
plt.figure(figsize=(12, 4))
for i in range(top_N):
    plt.subplot(1, top_N, i + 1)
    plt.imshow(top5_hardest_images.iloc[i].values.reshape(16, 16), cmap='gray')
    plt.title(f"Label: {top5_hardest_labels.iloc[i]}")
    plt.axis('off')

plt.show()
```

0.0.5 3.5: Gaussian Kernel

We now consider a Gaussian kernel function instead of a polynomial kernel function. We now deal with a parameter c as the width of the kernel. After initial experiments of trying $c = \{0.001, 0.01, 0.1, 1, 10, 100\}$ to identify the optimal magnitude of c to minimise the test error, we identify $c = 0.01$ as a suitable magnitude. Hence we select the range of c values S, to be $[0.005, 0.01, 0.02, 0.03, 0.04]$. We first average training and test errors over 20 runs for each value of $c \in S$.

c	**Train Error** $(\mu \pm \sigma)$ in %	**Test Error** $(\mu \pm \sigma)$ in %
0.005	0.1627 ± 0.0835	3.3602 ± 0.4170
0.01	0.0565 ± 0.0426	3.0591 ± 0.4514
0.02	0.0457 ± 0.0316	3.0108 ± 0.3713
0.03	0.0175 ± 0.0181	3.3548 ± 0.5016
0.04	0.0188 ± 0.0210	3.5645 ± 0.3332

This table highlights the errors for each value of c from our chosen S, which was centered around an initial run to identify magnitude. We obtain very strong and similar values for the test error, for an optimal c of $c^* = 0.02$. However once again, the lack of robustness/confidence in this value due to the very similar test errors for surrounding values of c, motivate the need for 5-fold cross-validation.

Mean c^* with std	**Test Error** $(\mu \pm \sigma)$ in %
0.016 ± 0.0450	3.0257 ± 0.5012

[ ]:
```
def question3_5(model, c_widths, df, confusion_matrix_print: bool,␣
 ↪misclassified_points: bool, num_runs=20):
    results = {c: {'train_errors': [], 'test_errors': []} for c in c_widths}
    for c in c_widths:
```

```
        print('c =', c)
        model.gaussian_gram_matrix(c)
        for run in range(num_runs):
            X_train, y_train, X_test, y_test = train_test_split(df)
            # Train perceptrons for the current width (c) and calculate error␣
 ↪rates
            train_error = model.fit(X_train, y_train)
            test_error = model.evaluate(X_train, X_test, y_test,␣
 ↪confusion_matrix_print, misclassified_points)

            results[c]['train_errors'].append(train_error)
            results[c]['test_errors'].append(test_error)
    return results
```

[ ]:
```
model = KPOVr(10, 15, True, 0.001)
c_widths = [0.005, 0.01, 0.02, 0.03, 0.04]
results3_5 = question3_5(model, c_widths, df, False, False, num_runs=20)

for c in c_widths:
    train_mean = np.mean(results3_5[c]['train_errors'])
    train_std = np.std(results3_5[c]['train_errors'])
    test_mean = np.mean(results3_5[c]['test_errors'])
    test_std = np.std(results3_5[c]['test_errors'])

    print(f"c = {c}: Train Error = {train_mean:.4f} ± {train_std:.4f}, Test␣
 ↪Error = {test_mean:.4f} ± {test_std:.4f}")
```

We then perform 5-fold cross-validation to identify the optimal c^* over 20 runs.

[ ]:
```
def question3_5_cv(model, c_widths, df, confusion_matrix_print: bool,␣
 ↪misclassified_points: bool, num_runs=20, num_folds=5):
    best_cs = []  # To store the best c for each run
    test_errors = []  # To store the test errors for each run
    confusion_matrices = []  # To store the confusion matrices for each run

    for run in range(num_runs):
        print(f"Run {run + 1}")
        X_train, y_train, X_test, y_test = train_test_split(df)
        fold_size = len(X_train) // num_folds
        best_c = None
        best_validation_error = float('inf')

        for c in c_widths:
            model.gaussian_gram_matrix(c)
            validation_errors = []

            for fold in range(num_folds):
```

```
                start = fold * fold_size
                end = (fold + 1) * fold_size
                X_fold = X_train.iloc[start:end]
                y_fold = y_train.iloc[start:end]
                mask = ~X_train.index.isin(range(start, end))
                X_remainder = X_train[mask]
                y_remainder = y_train[mask]

                model.fit(X_remainder, y_remainder)
                validation_error = model.evaluate(X_remainder, X_fold, y_fold,␣
 ↪False, False)
                validation_errors.append(validation_error)

            mean_validation_error = np.mean(validation_errors)
            if mean_validation_error < best_validation_error:
                best_validation_error = mean_validation_error
                best_c = c

        # Retrain with the best c on the full 80% training set
        model.gaussian_gram_matrix(best_c)
        model.fit(X_train, y_train)
        test_error = model.evaluate(X_train, X_test, y_test,␣
 ↪confusion_matrix_print, misclassified_points)
        print(f"Best c: {best_c}, Test Error: {test_error:.4f}\n")

        best_cs.append(best_c)
        test_errors.append(test_error)

    return best_cs, test_errors
```

[ ]:
```
# Perform cross-validation
model = KPOVr(10, 15, True, 0.001)
c_widths = [0.005, 0.01, 0.02, 0.03, 0.04]
best_cs, test_errors = question3_5_cv(model, c_widths, df, False, False,␣
 ↪num_runs=1, num_folds=5)

mean_best_c = np.mean(best_cs)
std_best_c = np.std(best_cs)
mean_test_error = np.mean(test_errors)
std_test_error = np.std(test_errors)

print(f"Mean Best c: {mean_best_c:.4f} ± {std_best_c:.4f}")
print(f"Mean Test Error: {mean_test_error:.4f} ± {std_test_error:.4f}")
```

0.0.6 3.6: Alternate Method: One-vs-One

Now we employ a One-vs-One approach, as opposed to One-vs-Rest, to generalise the kernel perceptron to k-classes. This is computationally more expensive, however allows for direct comparisons between classes and may prevent the classifier from predicting a image to be multiple classes with high probability.

d	**Train Error** $(\mu \pm \sigma)$ in %	**Test Error** $(\mu \pm \sigma)$ in %
1	1.6364 ± 0.0493	0.3538 ± 0.0823
2	0.0542 ± 0.0132	0.3201 ± 0.0733
3	0.0247 ± 0.0097	0.2771 ± 0.0659
4	0.0167 ± 0.0106	0.2596 ± 0.0616
5	0.0124 ± 0.0108	0.2512 ± 0.0723
6	0.0085 ± 0.0072	0.2612 ± 0.0545
7	0.0122 ± 0.0083	0.2671 ± 0.0560

Here we observe slightly lower test errors compared to One-vs-Rest, with lower standard deviations as well. This indicates a strong model, however once again there is ambiguity over the optimal d^*, so we perform 5-fold cross validation with the following results:

Mean d^* with std	**Test Error** $(\mu \pm \sigma)$ in %
4.3 ± 0.714	0.2623 ± 0.0393

After 5-fold cross validation, we obtain an optimal d^* which is also lower than that of One-vs-Rest, perhaps since there is less complexity within a decision boundary between 2 digits, as opposed to differentiating between 1 and 9 digits.

[ ]:
```
class KPOVo(KernelPerceptron):
    def __init__(self, num_classes: int, max_epochs: int, shuffle: bool,␣
 ↪epsilon: float):
        self.num_classes = num_classes
        self.max_epochs = max(max_epochs, 10)
        self.shuffle = shuffle
        self.epsilon = epsilon
        self.alpha = None
        self.kernel_func = None

    def fit(self, X_train, y_train):
        gram_matrix = self.kernel_func(X_train.values, X_train.values)
        self.alpha = np.zeros((self.num_classes, self.num_classes, X_train.
 ↪shape[0]))
        train_errors = []

        for i in range(self.max_epochs):
            train_mistakes = 0
```

```
            if self.shuffle:
                indices = np.random.choice(X_train.shape[0], X_train.shape[0],␣
↪replace=False)
            else:
                indices = np.arange(0, X_train.shape[0], 1)

            for example in indices:
                for class1 in range(self.num_classes):
                    for class2 in range(class1 + 1, self.num_classes):
                        y_class1 = self.alpha[class1, class2] @ gram_matrix[:,␣
↪example]
                        y_class2 = self.alpha[class2, class1] @ gram_matrix[:,␣
↪example]

                        if y_train.values[example] == class1:
                            if y_class1 <= y_class2:
                                self.alpha[class1, class2, example] += 1
                                train_mistakes += 1
                        else:
                            if y_class2 <= y_class1:
                                self.alpha[class2, class1, example] += 1
                                train_mistakes += 1

            train_error = train_mistakes / (X_train.shape[0] * self.num_classes␣
↪* (self.num_classes - 1) / 2) * 100
            train_errors.append(train_error)

            if i > 10:
                if np.abs(np.mean(train_errors[i - 5:i]) - np.
↪mean(train_errors[i - 4:i])) < self.epsilon:
                    return train_errors[-1]

        return train_errors[-1]

    def evaluate(self, X_train, X_test, y_test):
        gram_matrix = self.kernel_func(X_train.values, X_test.values)
        num_test_samples = X_test.shape[0]
        num_combinations = self.num_classes * (self.num_classes - 1) // 2

        class_pairs = [(class1, class2) for class1 in range(self.num_classes)␣
↪for class2 in range(class1 + 1, self.num_classes)]
        y_hat_labels = np.zeros((num_combinations, num_test_samples))

        for idx, (class1, class2) in enumerate(class_pairs):
            y_class1 = self.alpha[class1, class2] @ gram_matrix
            y_class2 = self.alpha[class2, class1] @ gram_matrix
```

```
            y_hat_labels[idx] = (y_class1 <= y_class2)

        y_test_matrix = np.zeros((num_combinations, num_test_samples))
        for idx, (class1, class2) in enumerate(class_pairs):
            y_test_matrix[idx] = (y_test.values == class1) | (y_test.values ==␣
↪class2)

        test_error = np.mean(np.sum(y_hat_labels != y_test_matrix, axis=0) > 0)␣
↪/ num_test_samples * 100

        return test_error

    def poly_gram_matrix(self, d):
        def kernel_func(x, y):
            return (x @ y.T) ** d
        self.kernel_func = kernel_func
```

[ ]:
```
def question3_6(model, degrees, df, num_runs=20):
    results = {d: {'train_errors': [], 'test_errors': []} for d in degrees}
    for d in degrees:
        print(f"d = {d}")
        model.poly_gram_matrix(d)
        for run in range(num_runs):
            print(f"run = {run+1}")
            X_train, y_train, X_test, y_test = train_test_split(df)
            # Train perceptrons for the current degree (d) and calculate error␣
↪rates
            train_error = model.fit(X_train, y_train)
            test_error = model.evaluate(X_train, X_test, y_test)
            # print('train_error =', train_error, '; test_error =', test_error)
            results[d]['train_errors'].append(train_error)
            results[d]['test_errors'].append(test_error)
    return results
```

[ ]:
```
model = KPOVo(10, 15, True, 0.001)
degrees = range(1,8)
results3_6 = question3_6(model, degrees, df, num_runs=20)

for d in degrees:
    train_mean = np.mean(results3_6[d]['train_errors'])
    train_std = np.std(results3_6[d]['train_errors'])
    test_mean = np.mean(results3_6[d]['test_errors'])
    test_std = np.std(results3_6[d]['test_errors'])

    print(f"d = {d}: Train Error = {train_mean:.4f} ± {train_std:.4f}, Test␣
↪Error = {test_mean:.4f} ± {test_std:.4f}")
```

[ ]:
```
def question3_6_cv(model, degrees, df, num_runs=20, num_folds=5):
    best_ds = []  # To store the best d for each run
    test_errors = []  # To store the test errors for each run

    for run in range(num_runs):
        X_train, y_train, X_test, y_test = train_test_split(df)
        fold_size = len(X_train) // num_folds
        best_d = None
        best_validation_error = float('inf')

        for d in degrees:
            model.poly_gram_matrix(d)
            validation_errors = []

            for fold in range(num_folds):
                start = fold * fold_size
                end = (fold + 1) * fold_size
                X_fold = X_train.iloc[start:end]
                y_fold = y_train.iloc[start:end]
                mask = ~X_train.index.isin(range(start, end))
                X_remainder = X_train[mask]
                y_remainder = y_train[mask]

                model.fit(X_remainder, y_remainder)
                validation_error = model.evaluate(X_remainder, X_fold, y_fold)
                validation_errors.append(validation_error)

            mean_validation_error = np.mean(validation_errors)
            if mean_validation_error < best_validation_error:
                best_validation_error = mean_validation_error
                best_d = d

        # Retrain with the best d on the full 80% training set
        model.poly_gram_matrix(best_d)
        model.fit(X_train, y_train)
        test_error = model.evaluate(X_train, X_test, y_test)
        print(f"Best d: {best_d}, Test Error: {test_error:.4f}\n")
        best_ds.append(best_d)
        test_errors.append(test_error)

    return best_ds, test_errors
```

[ ]:
```
# Perform cross-validation
model = KPOVo(10, 15, True, 0.001)
degrees = range(1, 8)
best_ds, test_errors = question3_6_cv(model, degrees, df, num_runs=20,␣
 ↪num_folds=5)
```

```
mean_best_d = np.mean(best_ds)
std_best_d = np.std(best_ds)
mean_test_error = np.mean(test_errors)
std_test_error = np.std(test_errors)

print(f"Mean Best d: {mean_best_d:.4f} ± {std_best_d:.4f}")
print(f"Mean Test Error: {mean_test_error:.4f} ± {std_test_error:.4f}")
```

0.0.7 Discussions

3.A Regarding parameters which were not cross-validated over, we identify a few which were not explored by choice. We could have identified the optimal maximum number of epochs "max_epochs", which may differ per kernel or dataset. However since the obtained test errors were sufficiently low, there was little potential gain from increasing the number of epochs and risk overfitting. In addition, epsilon as the convergence threshold could have been optimised from being fixed at 0.001. However, in a similar nature, the test errors were sufficiently low and there was little to gain from extending the training time by potentially decreasing epsilon. The number of classifiers remains trivially unchanged as contextually we are aware of exactly 10 classes, by the total number of unique labels in the data set (and human knowledge of our 10 digits).

3.B Initially we opted for the One-vs-Rest (OvR) approach where we train k separate binary classifiers, one for each class. Each classifier treats one class as positive and the rest as negative. This has advantages since it is computationally efficient, as we only create k classifiers for the k classes. One potential issue involves non-transitive comparisons, where a particular image might be classified as multiple classes with high confidence.

The other approach involves One-vs-One (OvO), where a binary classifier is trained for every pair of classes. With k classes, we will have $\frac{1}{2}k(k-1)$ classifiers, which is of course more computationally expensive compared to OvA. However, we do remove the issue of non-transitive comparisons since we distinguish between only 2 classes, rather than 1 class distinguished against $k-1$ classes.

In terms of our results, we find the OvO had a slightly better performance than OvA in terms of both mean and std testing errors, and also showed a slightly lower d^* for the polynomial kernel. At first glance, this could be due to an imbalanced dataset, as OvO tends to be less sensitive to this issue, however across the entire dataset, it is mostly balanced with the mode digit appearing ~1500 times, and the least frequent appearing ~700 times. There is some imbalance, but still heuristically one can argue that there is sufficient examples of all digits to not mean OvO has a huge improvement. Given the significantly longer computation time for OvO compared to OvR, it becomes more difficult to conclude on a better algorithm.

3.C Initially we trained with the polynomial kernel, with degree parameter d representing the polynomial degree. Higher degree polynomials can capture more complex relationships in the data, however we want to balance this against too high of a degree which could lead to overfitting. The Gaussian kernel with width parameter c controlling the kernel's width. Smaller c values lead to a wider kernel, which may cause underfitting, however if c is too small then we may overfit. The Gaussian kernel is more computationally expensive compared to the polynomial kernel, as it involving computing Euclidean distances. Gaussian kernels more robust and less susceptible to

noise.

In terms of our results, both kernels appeared to produce similar test errors at their optimal parameters (~3% test error rate), however the gaussian kernel took significantly longer to compute.

3.D In the implementation for OvR from 3.1, we discuss the sum $w(\cdot) = \sum_{i=0}^{m} \alpha_i K(x_i, \cdot)$ and how new terms are added to the sum during training. The sum of w was represented by a matrix multiplication between the alpha row vector α and each column of the gram matrix K, computing the weighted sum of kernel evalutations for each training image. These images were shuffled before each training run, with the algorithm operate on a single example at a time, hence administering an online learning setting. If a mistake is made i.e. the predicted class label does not match the true class label, then the alpha value corresponding to the true class label increases and simulataneously, we decrease that of the predicted class label. This happens for each image, hence continuously adding or subtracting terms to the sum during training.