Contents

Problem 1

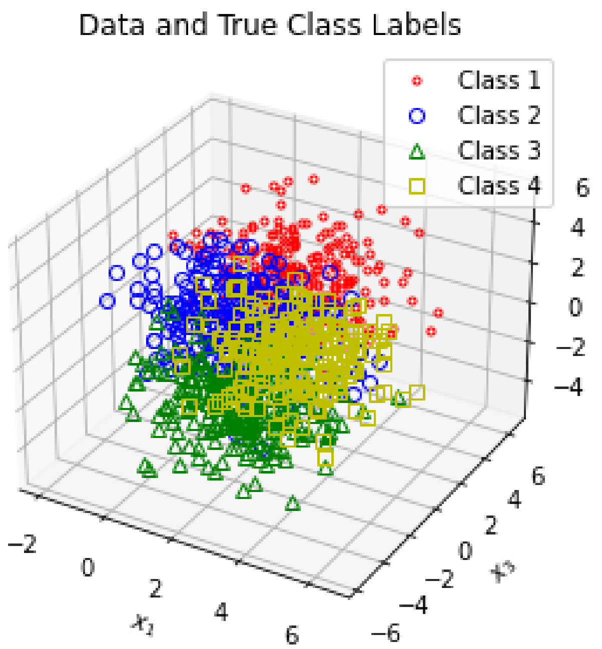
In this problem, several multilayer perceptrons (MLPs) were trained and used to approximate class label posteriors using the minimum average cross-entropy. These trained models were then used to estimate the minimum probability of error using the MAP classification rule on a validation dataset from cross-validation.

Data Distribution, Generation, and Theoretical min *P(Error)*

A 3-dimensional real-valued random vector X was generated from 4 classes with uniform priors and Gaussian class-conditional pdfs with the following means and covariances (all the covariances are equivalent):

$$\mu s = \begin{bmatrix}2\\2\\2\end{bmatrix}, \begin{bmatrix}2\\-2\\2\end{bmatrix}, \begin{bmatrix}2\\-2\\-2\end{bmatrix}, \begin{bmatrix}2\\2\\-2\end{bmatrix} \quad \Sigma s = \begin{bmatrix}2&0&0\\0&2&0\\0&0&2\end{bmatrix}$$

The resulting data distribution is displayed below:



These Gaussian pdfs were selected such that the theoretically optimal $P(Error)$ classifier, which utilizes knowledge of the true data pdf, achieves roughly a 10%-20% probability of error. The Maximum a Posteriori (MAP) classifier was used as the ERM classifier, resulting in a theoretically minimum probability of misclassification of 15.1%.

Using the specified gaussian pdf, training datasets were generated at 100, 200, 500, 1000, 2000, and 5000 samples with a test dataset of 100,000 samples.

MLP Structure

A 2-layer MLP with one hidden and one output layer was implemented with *P* perceptrons in the hidden layer with a ramp-style activation function (ReLU). The output layer used a "softmax" function, which ensures that all outputs are positive and sum to 1.

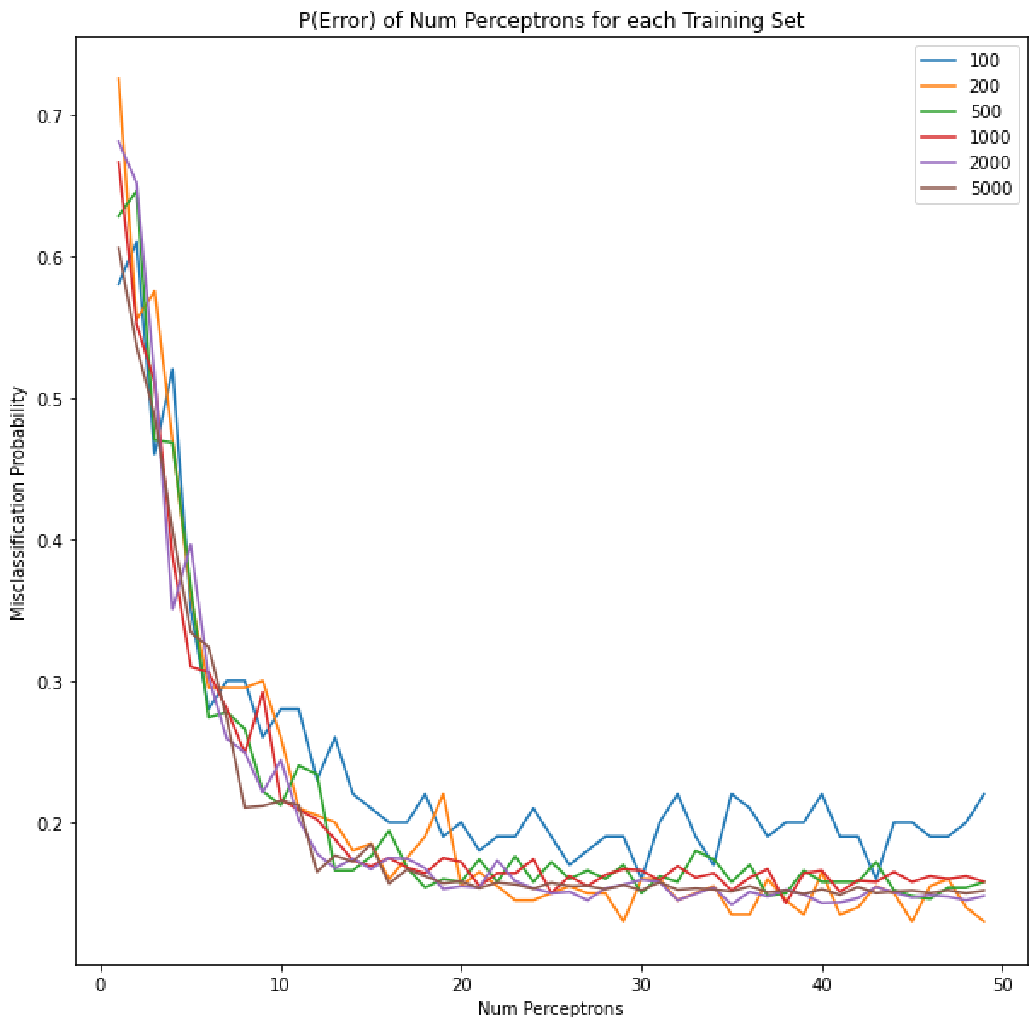
The hyperparameter *P,* representing the number of perceptrons in the first hidden layer, will be selected using cross-validation in the following sections.

Model Order Selection

For each training dataset, 10-fold cross validation was performed to find the optimal number of perceptrons for the MLP model. The objective function used in the hyperparameter selection was the minimum classification error probability. This procedure was completed as follows:

1. Split the data into 10 folds
2. Loop through 10 times using the fold with index i as the validation set with the other 9 folds being combined to form the training set.
3. Separate the data and labels of the obtained train and test datasets and return X_{train}, X_{test}, y_{train}, y_{test}
4. Create the two-layered MLP model
5. Define the optimizer (Stochastic Gradient Descent) and criterion (Cross Entropy Loss)
6. Train the model using the cross-validation training sets
7. Make predictions using the trained model on the validation data set
8. Use the minimum probability of error, $\min P(error)$, to evaluate performance for each fold
9. Average the $\min P(error)$ across all 10 folds – this $\min P(error)$ is the final evaluation metric
10. Repeat steps for each value of P (number of Perceptrons)
11. Find the minimum averaged $\min P(error)$ value and use the corresponding *P* as the optimal number of perceptrons.

The number of perceptrons was plotted against the $\min P(error)$ for each training dataset below. From it we can see that as the number of perceptrons increase, in every situation, the minimum probability decreases. Additionally, as the number of training samples increases, the predictability and smoothness of the plot increases. This can be seen in the jaggedness of the blue line (100 samples) compared to that of the brown line (5000 samples).



The ideal selected perceptron is not necessarily the one that minimizes the probability of error, rather it is the minimum number of perceptrons that does not result in decreased accuracy. In other words, the selected perceptron should be at the "elbow" of the plot above, where the probability of error levels off. This will decrease the complexity of the network, which is important for practical applications.

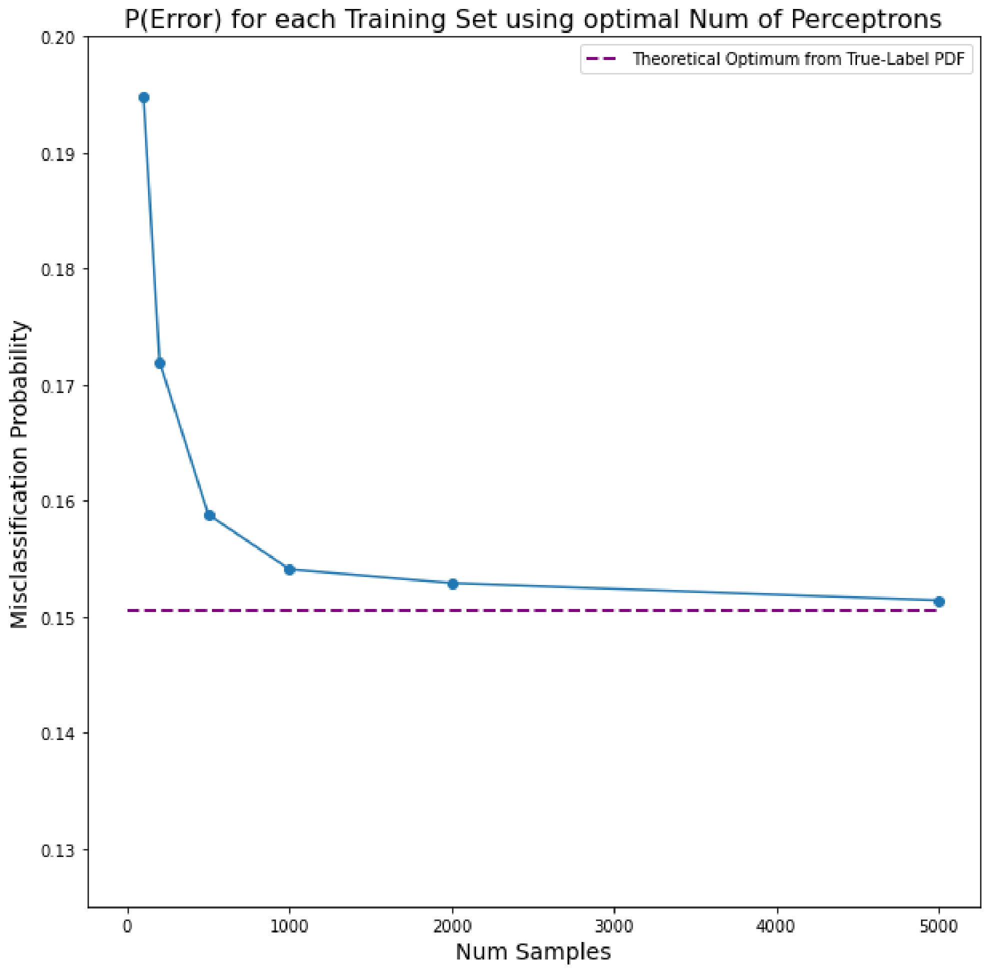
Model Training and Performance Assessment

The selected optimal number of perceptrons was then used to train the entire training dataset, resulting in the final trained MLP model. These MLP models for class posteriors were trained multiple times with random initialization to mitigate the chances of getting stuck at a local optimum. The highest training-data log-likelihood solution was used as the final MLP model.

Overall performance was analyzed by using these final trained models on the large test dataset to evaluate the model's ability to generalize. The samples in the test set were classified for each trained MLP and the probability of error was estimated empirically. The results are tabulated below.

Theoretical Optimum	100 Samples	200 Samples	500 Samples	1000 Samples	2000 Samples	5000 Samples
0.15019	0.18198	0.17654	0.15783	0.15539	0.15323	0.15131

A plot of the empirically estimated test probability of error for each trained MLP versus the number of training samples used in optimizing it is plotted below. The horizontal line indicates the empirically estimated test probability of error for the theoretically optimal classifier.



The overall probability of error is correlated with the size of the training dataset. As the size of the dataset increases, the probability of error decreases and approaches the minimum probability of error as estimated using the true pdf earlier in this problem.

Problem 2

The solution for ridge regression was derived in the case of a linear model with additive white Gaussian noise corrupting both input and output data. The solution was then implemented along with cross validation to select the hyperparameters.

Generate Data

The following sets were generated for this problem, where $n = 10, N_{train} = 50, and\ N_{test} = 1000$:

- $\boldsymbol{a}$ is an arbitrary non-zero n-dimensional random vector
- $\boldsymbol{x}$ is a $(N \times n)$ sampled from a multivariate gaussian pdf with non-zero mean vector and non-diagonal covariance matrix
    - Generated for both N_{train} and N_{test}
- $\boldsymbol{z}$ is a $(N \times n)$ sampled from a multivariate gaussian pdf with 0-mean and αI-covariance
    - Generated for both N_{train} and N_{test}
- $\boldsymbol{v}$ is a scalar N-dimensional random vector sampled from a 0-mean, unit-variance gaussian pdf.
    - Generated for both N_{train} and N_{test}
- $\boldsymbol{y}$ is the scalar N-dimensional output random vector calculated as $\boldsymbol{y} = \boldsymbol{a}^T(\boldsymbol{x} + \boldsymbol{z}) + \boldsymbol{v}$
    - Generated for both N_{train} and N_{test} using corresponding $\boldsymbol{x}, \boldsymbol{z}, and\ \boldsymbol{v}.$

These results are a training and testing dataset consisting of (x, y) pairs.

Model Parameter Estimation

The following models are created under the assumption that (x, y) pairs follow a linear relationship $(y = w^T x + w_0 + v)$ and that we have no knowledge of the noise term $\boldsymbol{z}$. A 0-mean and βI-covariance-matrix Gaussian pdf is assumed as a prior for the model parameters $\boldsymbol{w}$, which includes the bias term w_0. The assumption of a diagonal covariance matrix makes this a Ridge Regression problem, from which a MAP estimator can be derived as follows.

$$\widehat{w}_{map} = \underset{w}{argmax} \log P(w|D) \rightarrow Bayes \rightarrow \underset{w}{argmax}[\log P(D|w) + \log(w) - \log P(D)]$$

Where:

$$\log(w) = \log \frac{1}{\sqrt{2\pi\gamma}} \exp\left(-\frac{w^T w}{2\gamma^2}\right) = -\frac{w^T w}{2\gamma^2}$$

$$\log P(D|w) = \log \prod_{n=1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_n - w^T x)^2}{2\sigma^2}\right) = \sum_{n=1}^{n} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{(y_n - w^T x)^2}{2\sigma^2}$$
$$= -\frac{1}{2\sigma^2} \sum_{n=1}^{n} (y_n - w^T x)^2$$

From there, the expression for the estimated parameter vector becomes:

$$\widehat{w}_{map} = \underset{w}{argmax}\left[-\frac{1}{2\sigma^2}\sum_{n=1}^{n}(y_n - w^T x)^2 - \frac{w^T w}{2\gamma^2}\right]$$
$$= \underset{w}{argmax}\sum_{n=1}^{n}[y_n^2 - 2y_n w^T x + w^T x x^T w] + \frac{\sigma^2}{\gamma^2}w^T w$$

Taking the partial derivative $\frac{\partial}{\partial w}$ and setting it equal to zero:

$$0 = \sum_{n=1}^{n} -2y_n x_n + 2\sum_{n=1}^{n} x_n x_n^T \widehat{w} + \frac{2\sigma^2}{\gamma^2}\widehat{w}$$

$$\therefore yx = \left(x^T x + \frac{\sigma^2}{\gamma^2}\right)\widehat{w}_{map} \rightarrow \widehat{w}_{map} = \left(x^T x + \frac{\sigma^2}{\gamma^2}\right)^{-1} xy$$

The Ridge-Regression MAP estimator equation used for this problem is:

$$\widehat{w}_{map} = \left(X^T X + \frac{\sigma_v^2}{\beta}I\right)^{-1} X^T y$$

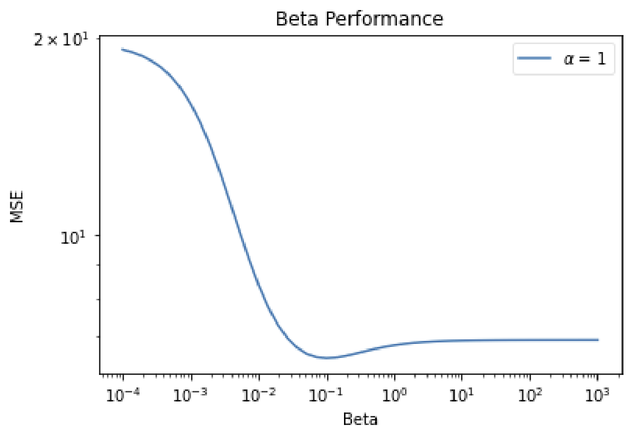
Where σ_v^2 is the variance of the model additive noise term v.

This MAP parameter estimation expression will be used to determine the optimal weights for the model using the training data while varying β and α.

Hyper-parameter Optimization

The prior being used for regularization has a scalar parameter β that will be selected using 5-fold cross-validation. The objective function for cross-validation used max-log-likelihood, which is equivalent to Residual Sum of Squares for a constant covariance. For conceptual simplicity, the Mean Squared Error (MSE) was used in visualization throughout. MSE is the normalized equivalent to RSS, averaging the loss values over the dataset (RSS/N samples). This leads to a more intuitive quantitative result.

Using a range of β values from 10^{-3} to 10^3, a parameter estimate vector $\boldsymbol{w}$ was found by using MAP with the training folds and compared against the MSE. The curve generated by plotting the hyperparameter β against the corresponding MSE has a minimum value that will be used as the optimal value for the following section.

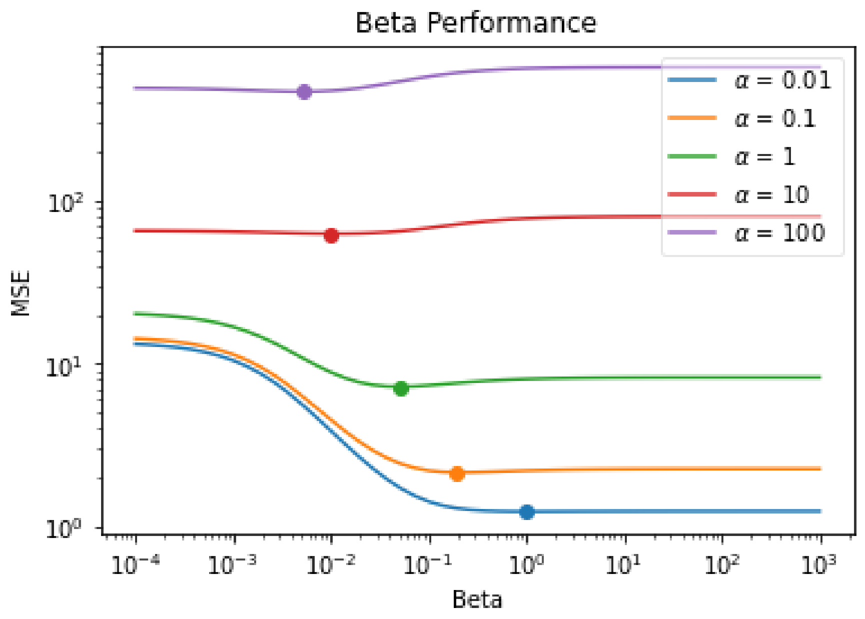


Model Optimization

Now that the optimal hyperparameter value of β has been identified, the entire training dataset was used to optimize the model parameters with the MAP parameter estimator. The performance of this model was analyzed using the test data. Once again, for visualization, the MSE was used for visualization, but it was also evaluated using the '-2 times log likelihood' of the test data, which is similar to finding the Negative Log-Likelihood estimate (NLL).

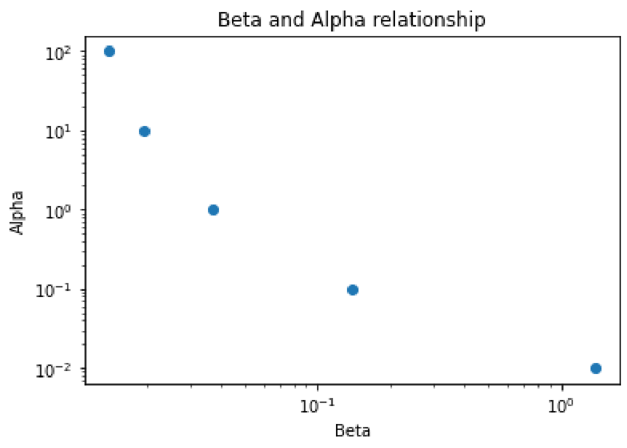
While keeping $\boldsymbol{a}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$ constant, the noise parameter α was swept from values of $10^{-3} trace(\Sigma)/n$ to $10^{3} trace(\Sigma)/n$. This introduces increasing levels of noise to the input variable. New $\boldsymbol{z}$ and $\boldsymbol{y}$ sets were generated using each value of α and the previous processes were repeated. This results in several curves for optimal β hyperparameters with varying α hyperparameters. The sweep of alpha was reduced for visualization, focusing on $\alpha = 0.01, 0.1, 1, 10, and\ 100$.

The figure below displays the average MSE computed during training for each of the β values evaluated. Each trace represents a different value of α and the optimal β value is that which minimizes the MSE, which are visualized as points on the plot.

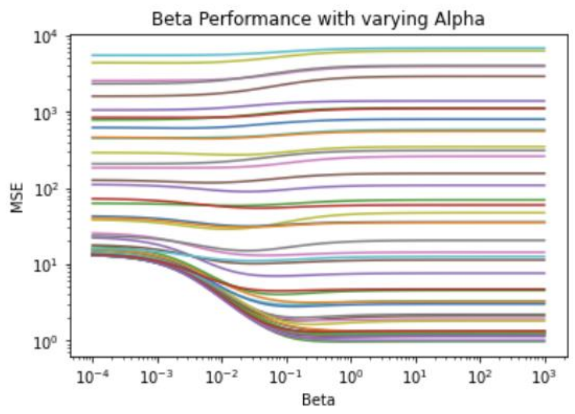


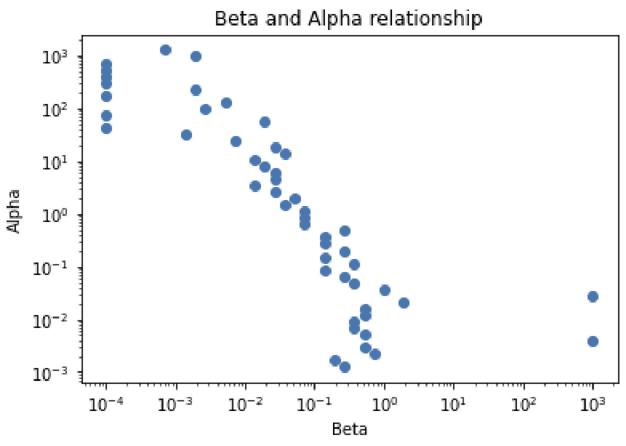
This shows that the optimal β-value is inversely proportional to the additive noise parameter α. This is expected, intuitively, because β is the inverse weight on the prior. Therefore, low values result in a reduced weighting of prior knowledge (MAP) and more weighting on the generated best fit estimate. Increasing the noise level results in a lower accuracy in the best fit estimate, leading to a higher importance on the prior to create a more accurate parameter estimate.

In other words, low α-values represent lower noise level, resulting in more reliable best fit approximation of the parameter vector. This results in a decreased importance on the prior, whose weight can be decreased by increasing the value of β. Similarly, as the noise level increases, more weight should be placed on the priors, so as α increases, β should decrease. This trend can be seen below for the same reduced sweep of alpha used for visualization.



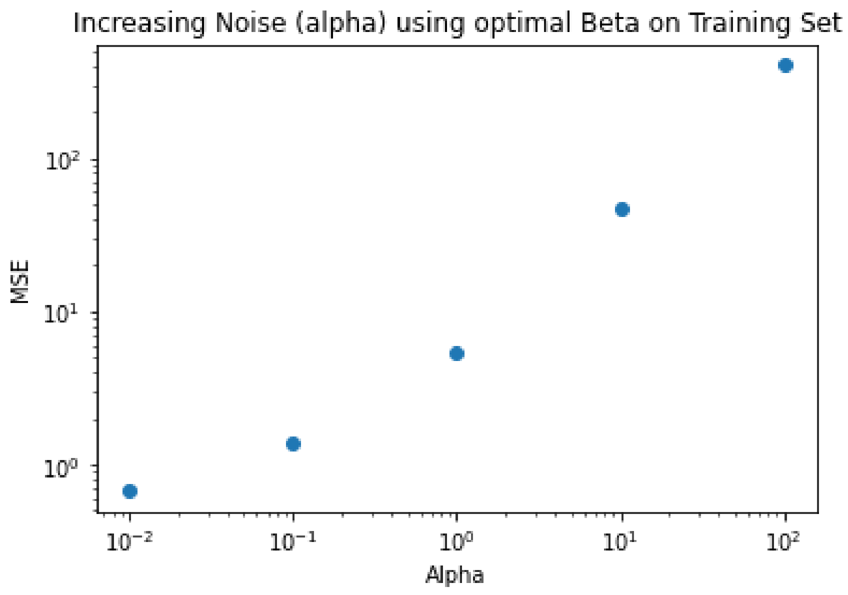
The plots using the full sweep of α can be seen below:

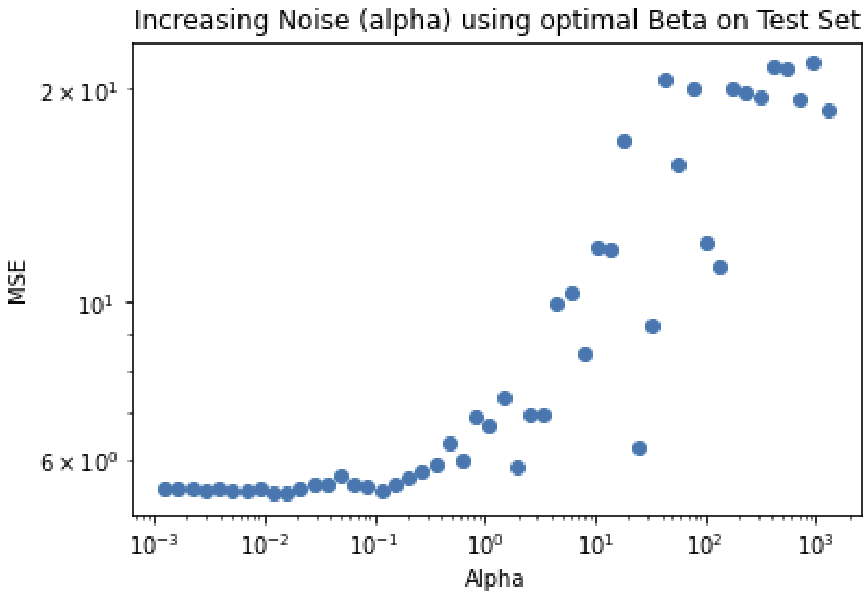




Performance Analysis

Finally, the optimal β-value was used to generate a final estimate using the whole training dataset for each value in the sweep of α. This model was then used on the test dataset, and the MSE was calculated, representing the final evaluation metric. In the plot below, as the noise level α increases, the MSE increases exponentially while also becoming smore noisy and unpredictable. The first plot is the reduced values of alpha, and the second plot is for the full sweep.


Increasing Noise (alpha) using optimal Beta on Training Set
MSE
Alpha
10^2
10^1
10^0
10^{-2}
10^{-1}
10^0
10^1
10^2


Increasing Noise (alpha) using optimal Beta on Test Set
MSE
Alpha
2×10^1
10^1
6×10^0
10^{-3}
10^{-2}
10^{-1}
10^0
10^1
10^2
10^3

Appendix

The Code for each problem can be found in the Appendices below.

It can also be found in my github at https://github.com/meuliano/Machine-Learning/tree/main/HW3 with the following python scripts:

- Problem 1: Euliano_eece5644_hw3_1.py
- Problem 2: Euliano_eece5644_hw3_2.py

Appendix A: Problem 1 Code

```
# %%
# Imports
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sys import float_info # Threshold smallest positive floating value
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import confusion_matrix

import pandas as pd

import torch
import torch.nn as nn

# %%
def generate_data_from_gmm(N, pdf_params):
    # Generate Data Function - Returns N x 3 and labels
    # Determine dimensionality from mixture PDF parameters
    n = pdf_params['m'].shape[1]
    # Output samples and labels
    X = np.zeros([N, n])
    labels = np.zeros(N)

    # Decide randomly which samples will come from each component
    u = np.random.rand(N)
    thresholds = np.cumsum(pdf_params['priors'])
    thresholds = np.insert(thresholds, 0, 0) # For intervals of classes

    L = np.array(range(1, len(pdf_params['priors'])+1))
    for l in L:
        # Get randomly sampled indices for this component
        indices = np.argwhere((thresholds[l-1] <= u) & (u <=
thresholds[l]))[:, 0]
        # No. of samples in this component
        Nl = len(indices)
        labels[indices] = l * np.ones(Nl) - 1
        if n == 1:
            X[indices, 0] =  norm.rvs(pdf_params['m'][l-1],
pdf_params['C'][l-1], Nl)
        else:
            X[indices, :] =  multivariate_normal.rvs(pdf_params['m'][l-1],
pdf_params['C'][l-1], Nl)

    return X, labels
```

```
# %%
def perform_erm_classification(X, Lambda, gmm_params, C):
    # ERM classification rule (min prob. of error classifier)
    # Conditional likelihoods of each x given each class, shape (C, N)
    class_cond_likelihoods = np.array([multivariate_normal.pdf(X,
gmm_params['m'][c], gmm_params['C'][c]) for c in range(C)])

    # Take diag so we have (C, C) shape of priors with prior prob along
diagonal
    class_priors = np.diag(gmm_params['priors'])
    # class_priors*likelihood with diagonal matrix creates a matrix of
posterior probabilities
    # with each class as a row and N columns for samples, e.g. row 1:
[p(y1)p(x1|y1), ..., p(y1)p(xN|y1)]
    class_posteriors = class_priors.dot(class_cond_likelihoods)

    # Conditional risk matrix of size C x N with each class as a row and N
columns for samples
    risk_mat = Lambda.dot(class_posteriors)

    return np.argmin(risk_mat, axis=0)
# %%
def train_model(model, data, labels, criterion, optimizer, num_epochs=25):
    # Set up training data
    X_train = torch.FloatTensor(data)
    y_train = torch.LongTensor(labels)

    # Optimize the neural network
    for epoch in range(num_epochs):
        # Set grads to zero explicitly before backprop
        optimizer.zero_grad()
        outputs = model(X_train)
        # Criterion computes the cross entropy loss between input and target
        loss = criterion(outputs, y_train)
        # Backward pass to compute the gradients through the network
        loss.backward()
        # GD step update
        optimizer.step()

    return model

# %%
def model_predict(model, data):
    # Set up test data as tensor
    X_test = torch.FloatTensor(data)

    # Evaluate nn on test data and compare to true labels
    predicted_labels = model(X_test)
    # Back to numpy
    predicted_labels = predicted_labels.detach().numpy()

    return np.argmax(predicted_labels, 1)

# %%
def mse(y_preds, y_true):
    # Residual error (X * theta) - y
    error = y_preds - y_true
    # Loss function is MSE
    return np.mean(error ** 2)

# %%
def model_order_selection(X_train, y_train, folds, poly_deg):

  C = len(np.unique(y_train))

  cv = KFold(n_splits=folds, shuffle=True)

  # Polynomial degrees ("hyperparameters") to evaluate
  degs = np.arange(1, poly_deg, 1)
  n_degs = np.max(degs)
  error_prob=np.empty((n_degs,folds))
  all_models = []
  #
  for deg in degs:
    k = 0

    # split training set for 10-fold cross validation
    # for each of the 10 smaller sets, train model for 1-10 perceptrons
    for fold, (train_indices, valid_indices) in enumerate(cv.split(X_train)):
        # Extract the training and validation sets from the K-fold split
        X_train_k = X_train[train_indices]
        y_train_k = y_train[train_indices]
        X_valid_k = X_train[valid_indices]
        y_valid_k = y_train[valid_indices]

        model = TwoLayerMLP(X_train.shape[1], deg, C)

        optimizer = torch.optim.SGD(model.parameters(), lr=0.001,
momentum=0.9)
        # The nn.CrossEntropyLoss() loss function automatically performs a
log_softmax() to
        # the output when validating, on top of calculating the negative-log-
likelihood using
        # nn.NLLLoss(), while also being more stable numerically... So don't
implement from scratch
        criterion = nn.CrossEntropyLoss()
        trained_model = train_model(model, X_train_k, y_train_k, criterion,
optimizer, num_epochs=200)
        all_models.append(trained_model)

        # Make predictions from validation data
        y_valid_pred = model_predict(trained_model,X_valid_k)
        num_errors = len(np.argwhere(y_valid_pred != y_valid_k))
        error_prob[deg-1,k] = num_errors/len(y_valid_k)
        k += 1


  error_prob_m = np.mean(error_prob, axis=1)

  # +1 as the index starts from 0 while the degrees start from 1
  optimal_d = np.argmin(error_prob_m) + 1
  #print("The model selected to best fit the data without overfitting is:
d={}".format(optimal_d))
  optimal_hit = error_prob_m[optimal_d-1]
  print("Perceptrons: ", optimal_d, "  Probability of Error: ",optimal_hit)

  return optimal_d, error_prob_m



# %%
class TwoLayerMLP(nn.Module):
    # The nn.CrossEntropyLoss() loss function automatically performs a
log_softmax() to
    # the output when validating, on top of calculating the negative-log-
likelihood using
    # nn.NLLLoss(), while also being more stable numerically... So don't
implement from scratch
    # https://pytorch.org/docs/stable/generated/torch.nn.Module.html

    def __init__(self, d_in, d_hidden, C):
        super(TwoLayerMLP, self).__init__()

        self.fc1 = nn.Linear(d_in, d_hidden)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(d_hidden, C)
        self.log_softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)  # fc to perceptrons
        x = self.relu(x) # or self.softplus(x) for smooth-ReLU, empirically
worse than ReLU
        x = self.fc2(x)  # connect to output layer
        x = self.log_softmax(x)  # for outputs that sum to 1
        return x

# %%
def main():
    # Set Gaussian PDF params, Generate Data, Plot the original data and
their true labels
    N = 1000
    n = 3 # dimensionality of input random vectors
    C = 4 # number of classes

    gmm_pdf = {}
    gmm_pdf['priors'] = np.array([1/C, 1/C, 1/C, 1/C])
    mu0 = [2,  2,  2]
    mu1 = [2, -2,  2]
    mu2 = [2, -2, -2]
    mu3 = [2,  2, -2]
    gmm_pdf['m'] = np.array([mu0, mu1, mu2, mu3])
    gmm_pdf['C'] = np.array([2*np.eye(n), 2*np.eye(n), 2*np.eye(n),
2*np.eye(n)])

    X, labels = generate_data_from_gmm(N, gmm_pdf)
    fig = plt.figure()
    ax_gmm = fig.add_subplot(111, projection='3d')

    ax_gmm.plot(X[labels == 0, 0], X[labels == 0, 1], X[labels == 0, 2],'r.',
label="Class 1", markerfacecolor='none')
```

```
    ax_gmm.plot(X[labels == 1, 0], X[labels == 1, 1], X[labels == 1, 2],'bo',
label="Class 2", markerfacecolor='none')
    ax_gmm.plot(X[labels == 2, 0], X[labels == 2, 1], X[labels == 2, 2],'g^',
label="Class 3", markerfacecolor='none')
    ax_gmm.plot(X[labels == 3, 0], X[labels == 3, 1], X[labels == 3, 2],'ys',
label="Class 4", markerfacecolor='none')
    ax_gmm.set_xlabel(r"$x_1$")
    ax_gmm.set_ylabel(r"$x_2$")
    ax_gmm.set_ylabel(r"$x_3$")

    plt.title("Data and True Class Labels")
    plt.legend()
    plt.tight_layout()
    plt.show()

    # %%
    # Calculate Theoretical Optimal P(Error)
    N = 100000
    X, labels = generate_data_from_gmm(N, gmm_pdf)

    # If 0-1 loss then yield MAP decision rule, else ERM classifier
    Lambda = np.ones((C, C)) - np.eye(C)

    # ERM decision rule, take index/label associated with minimum conditional
risk as decision (N, 1)
    decisions = perform_erm_classification(X, Lambda, gmm_pdf, C)

    num_errors = len(np.argwhere(decisions != labels))
    error_prob_th = num_errors/len(labels)
    print("Theoretically Optimal Misclassification Probability =
",error_prob_th)

    # %%
    X_test, y_test = generate_data_from_gmm(100000,gmm_pdf)

    # %%
    fig = plt.figure()

    degs = np.arange(1, 50, 1) # Reduce this to speed up

    N = [100,200,500,1000,2000,5000]
    err_prob_est = []

    for samples in N:
        X_train, y_train = generate_data_from_gmm(samples, gmm_pdf)
        op_d, err_prob_m = model_order_selection(X_train, y_train, folds=10,
poly_deg=len(degs)+1)
        plt.plot(degs,err_prob_m,label=samples)

        model = TwoLayerMLP(X_train.shape[1], op_d, C)
        optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
        criterion = nn.CrossEntropyLoss()
        trained_model = train_model(model, X_train, y_train, criterion,
optimizer, num_epochs=200)
        y_test_pred = model_predict(trained_model,X_test)
        num_errors = len(np.argwhere(y_test_pred != y_test))
        err_prob_est.append(num_errors/len(y_test))
        print("Misclassification Probability = ",err_prob_est)

    plt.title("P(Error) of Num Perceptrons for each Training Set")
    plt.legend()
    plt.xlabel("Num Perceptrons")
    plt.ylabel("Misclassification Probability")
    plt.show()
    #

    # %%
    fig = plt.figure()
    plt.scatter(N,err_prob_est)
    plt.plot(N,err_prob_est)
    plt.hlines(y=error_prob_th,xmin=0, xmax=samples,colors="purple",
linestyles='--', lw=2, label='Theoretical Optimum from True-Label PDF')
    plt.title("P(Error) for each Training Set using optimal Num of
Perceptrons", fontsize=16)
    plt.legend()
    plt.xscale('log')
    plt.ylim(0.125,0.2)
    plt.xlabel("Num Samples", fontsize=14)
    plt.ylabel("Misclassification Probability", fontsize=14)
    plt.show()


if __name__ == '__main__':
    main()
```

Appendix B: Problem 2 Code

```
# %%
# %%
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from sys import float_info # Threshold smallest positive floating value
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import confusion_matrix

import torch
import torch.nn as nn

# Set seed to generate reproducible "pseudo-randomness" (handles scipy's
"randomness" too)
np.random.seed(4)

# %%
def mapParamEstimate(x,y,param,sigma):
    # (xT * X + gamma * I)^-1 * xT * y
    x_tilde = np.vstack((np.ones(x.shape[0]), x.T))
    w_hat = np.linalg.inv(x_tilde.dot(x_tilde.T) +
sigma**2/param*np.eye(x_tilde.shape[0])).dot(x_tilde).dot(y)
    return w_hat

def meanSquaredError(w, x, y):
    N = len(y)
    x_tilde = np.vstack((np.ones(x.shape[0]), x.T))
    error = (y-x_tilde.T.dot(w))**2
    mse = np.sum(error)/N
    return mse

def main():
    # %%
    n = 10
    N_train = 50
    N_test = 1000

    # %%
    # Generate Data
    # Define pdfs for random variables
    #np.random.seed(4)
    # Draw N_train iid samples of n-dimensional samples of x
    a = np.array([np.random.rand(n)]) # arbitrary non-zero n-dim vector a
    x_mu = np.random.rand(n) # arbitrary non-zero mean
    sigma = np.eye(n) + 0.2*np.random.rand(n,n) # arbitrary non-diagonal
covariance
    x_sigma = sigma.T * sigma # Force positive Semi-definite

    # Define Alphas for parametric sweep
    alpha = 1
    #
    # Draw N_train iid samples of scalar random variable v - 0-mean, alpha-I
covariance
    z_mu = np.zeros(n)
    z_sigma = alpha*np.eye(n)
    # Draw N_train iid samples of random variable v from 0-mean, unit-
variance
    v_mu = 0
    v_sigma = 1


    # GENERATE DATA
    # Training Data
    x_train = multivariate_normal.rvs(x_mu,x_sigma,N_train)
    z_train = multivariate_normal.rvs(z_mu,z_sigma,N_train)
    v_train = v_mu+v_sigma*np.random.randn(N_train,1)

    # This is the training dataset
    y_train = (a.dot(x_train.T+z_train.T)).T + v_train


    # Test Data
    x_test = multivariate_normal.rvs(x_mu,x_sigma,N_test)
    z_test = multivariate_normal.rvs(z_mu,z_sigma,N_test)
    v_test = v_mu+v_sigma*np.random.randn(N_test,1)

    y_test = (a.dot(x_test.T+z_test.T)).T + v_test

    # %%
    # Use MAP Parameter estimation to determine the optimal weights for this
model
    betas = np.geomspace(10**-4,10**4,17)
    w_maps = np.array([mapParamEstimate(x_train,y_train,beta,v_sigma) for
beta in betas])
    mse_map = np.array([meanSquaredError(w_map, x_test, y_test) for w_map in
w_maps])
    print("Mean Squared Error - MAP Estimator:\n",mse_map)
    # print("Gamma Values:\n",gammas)
    print("condition: ", np.linalg.cond(x_train.T.dot(x_train)))

    # %%
    # Define Alphas for parametric sweep
    alpha = 1
    alpha_sweep = np.logspace(-3,3)*np.trace(x_sigma)/n
    # alpha_plot_ind = round(1:len(alpha_sweep)/(7:len(alpha_sweep)))
    #
    # Draw N_train iid samples of scalar random variable v - 0-mean, alpha-I
covariance
    z_mu = np.zeros(n)
    z_sigma = alpha*np.eye(n)
    z_train = multivariate_normal.rvs(z_mu,z_sigma,N_train)

    alpha_sweep = [0.01,0.1,1,10,100]
    # alpha_sweep = [1]
    y_train_sweep = []
    # Alpha Sweep Loop
    for alpha_ind in range(len(alpha_sweep)):
        z_sigma_sweep = alpha_sweep[alpha_ind]*np.eye(n)
        # z_sigmas.append(z_sigma_sweep)
        z_trains = multivariate_normal.rvs(z_mu,z_sigma_sweep,N_train)
        y_train_sweep.append((a.dot(x_train.T+z_trains.T)).T + v_train)
    # %%
    betas = np.logspace(-4,3) # 50 Beta vals to try from 1e-4 to 1e3
    cv = KFold(n_splits=5, shuffle=True) # Setup KFold with 5 folds
    mse_map = np.empty((50, 5)) # Allocate space for matrix of MSEs

    plt.figure()
    alpha_ind = 0
    beta_opts = []

    # Perform Cross-Validation
    for y_train in y_train_sweep:
        # y_train = (a.dot(x_train.T+z_train.T)).T + v_train

        for fold, (train_indices, valid_indices) in
enumerate(cv.split(x_train)):
            # Extract the training and validation sets from the K-fold split
            X_train_k = x_train[train_indices]
            y_train_k = y_train[train_indices]
            X_valid_k = x_train[valid_indices]
            y_valid_k = y_train[valid_indices]

            # Find Parameter Vector using MAP equation for each fold and for
each value of beta
            w_maps =
np.array([mapParamEstimate(X_train_k,y_train_k,beta,v_sigma) for beta in
betas])
            # Find the MSE for each generated param vector using validation
set
            mse_map[:,fold] = np.array([meanSquaredError(w_map, X_valid_k,
y_valid_k) for w_map in w_maps])

        # Average across each fold - results in MSE averaged estimate for
each Beta value
        avgs = [np.average(mse_map[i,:]) for i in range(mse_map.shape[0])]

        beta_opt = betas[avgs.index(np.min(avgs))]
        beta_opts.append(beta_opt)
        print("Optimal Beta = ", beta_opt)
        alpha_val = alpha_sweep[alpha_ind]



        # TURN THIS INTO TWO FIGURES!
        # plt.scatter(beta_opt,alpha_val)
        plt.scatter(beta_opt,np.min(avgs))
        plt.plot(betas,avgs,label=r"$\alpha$ = {}".format(alpha_val))
        alpha_ind = alpha_ind + 1

    plt.legend(loc='best', fancybox=True, framealpha=0.5)
    plt.title("Beta Performance")
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel("Beta")
    plt.ylabel("MSE")
    plt.show()
    # %%
    plt.scatter(beta_opts,alpha_sweep)
    plt.title("Beta and Alpha relationship")
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel("Beta")
    plt.ylabel("Alpha")
    plt.show()

    # %%
    # Now we have trained optimal weights for sweeps of Beta and Alpha
    # Use the entire training dataset to optimize the model parameters using
MAP using best hyperparams

    # Find Parameter Vector using MAP equation for each fold and for each
value of beta
    w_maps_train =
np.array([mapParamEstimate(x_train,y_train_sweep[i],beta_opts[i],v_sigma) for
i in range(len(beta_opts))])
    # Find the MSE for each generated param vector using validation set
    mse_map_train = np.array([meanSquaredError(w_maps_train[i], x_train,
y_train_sweep[i]) for i in range(w_maps_train.shape[0])])
    # -2 times log likelihood of TEST DATA ?????

    plt.scatter(alpha_sweep,mse_map_train)
    plt.title("Increasing Noise (alpha) using optimal Beta on Training Set")
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel("Alpha")
    plt.ylabel("MSE")
    plt.show()


    # %%
    mse_map_train = np.array([meanSquaredError(w_map, x_test, y_test) for
w_map in w_maps_train])
    # -2 times log likelihood of TEST DATA ?????

    plt.scatter(alpha_sweep,mse_map_train)
    plt.title("Increasing Noise (alpha) using optimal Beta on Test Set")
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel("Alpha")
    plt.ylabel("MSE")
    plt.show()

if __name__ == '__main__':
    main()
```