Q1

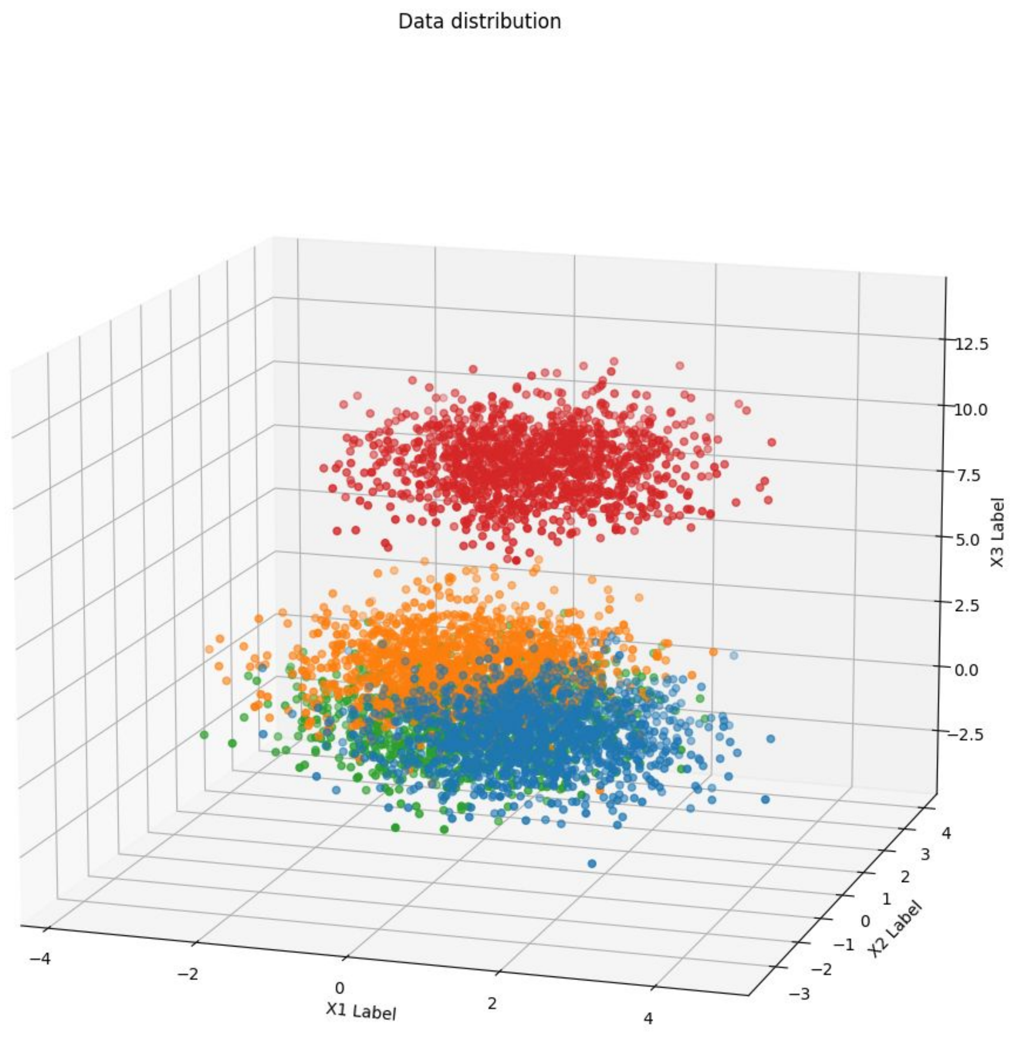
Chosen Parameters

Below are the chosen ***μ*** and **Σ** values chosen for the 4 gaussian class conditionals

$$_1 = [1\ 0\ 0]\quad {}_2 = [0\ 0\ 2.5]\quad {}_3 = [0\ 0.5\ 0]\quad {}_4 = [1\ 0\ 10]\,\backslash$$

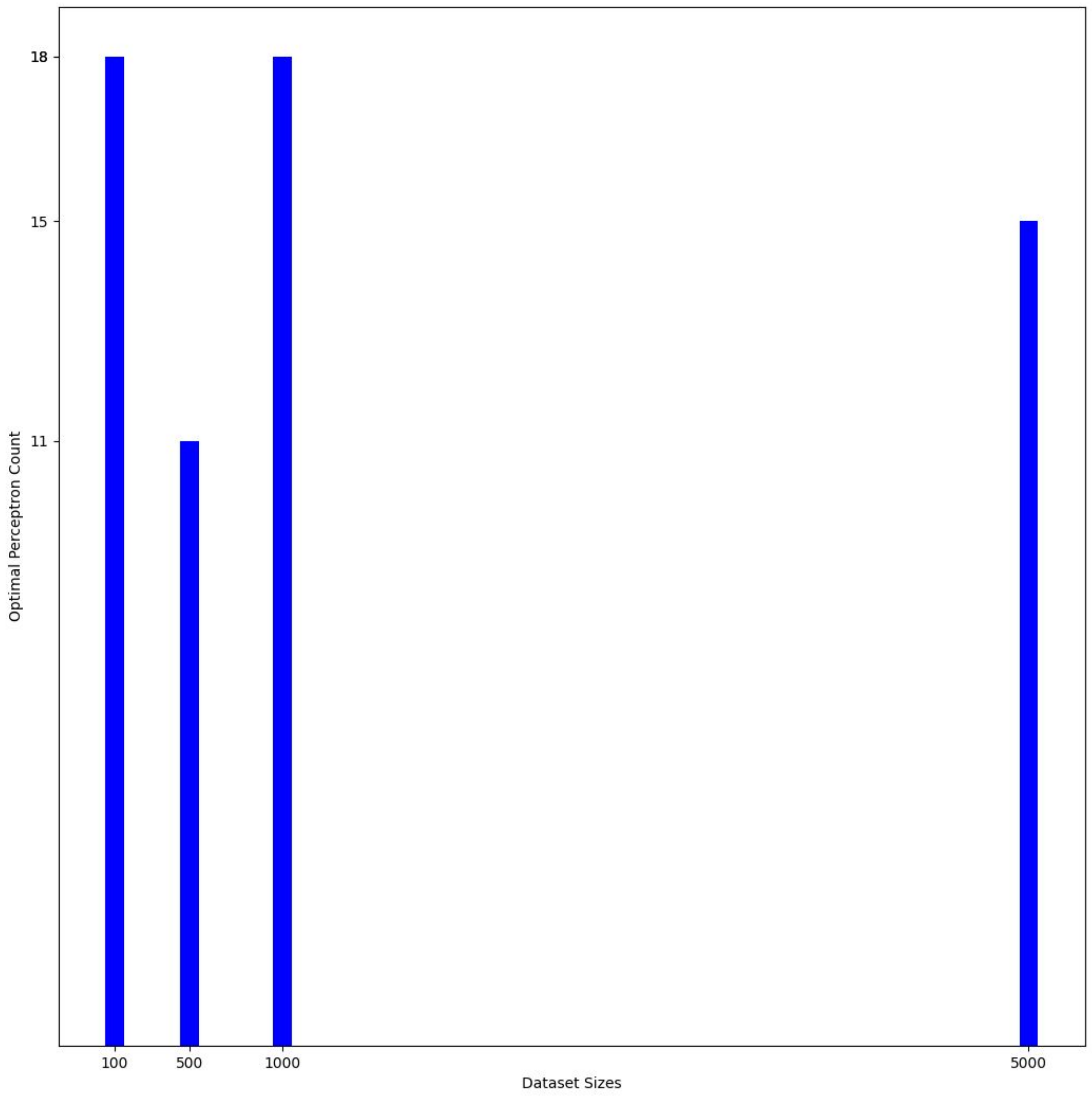
$$_1 = I^{3x3}\quad {}_2 = I^{3x3}\quad {}_3 = I^{3x3}\quad {}_4 = I^{3x3}$$

The below data distribution results in an optimal MAP classifier accuracy of ~80% using true knowledge of the data when the classifier is applied to validation set with 100K samples.



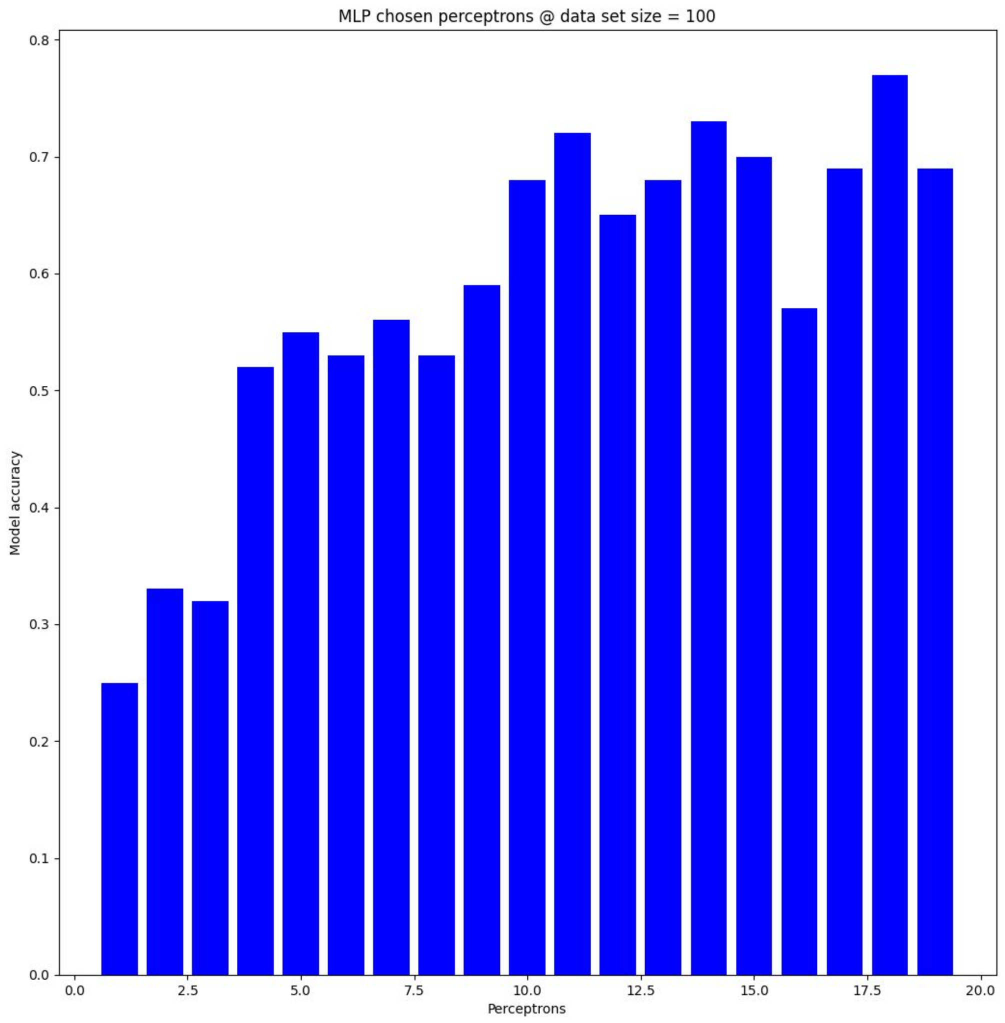
Kfold cross validation for perceptron selection

From the below figure, kfold generally favours higher perceptron counts at lower dataset sizes. At high dataset sizes it favours lower perceptron counts. Unfortunately multiple kfold runs weren't performed multiple times for each dataset, nor were enough datasets generated with kfold run on them to establish a clear trend. A violation of the above assumption is the optimally chosen 11 perceptrons for the 500 dataset. Observing the histograms of accuracy vs perceptron count for each dataset can shed more light on this unexpected behaviour. In the following figures, it will become apparent that the optimally selected perceptron counts rely on minute differences based on average trained model performance on validation partitions during kfold. These differences are so minute they can almost be disregarded which renders the below graph less useful than expected.



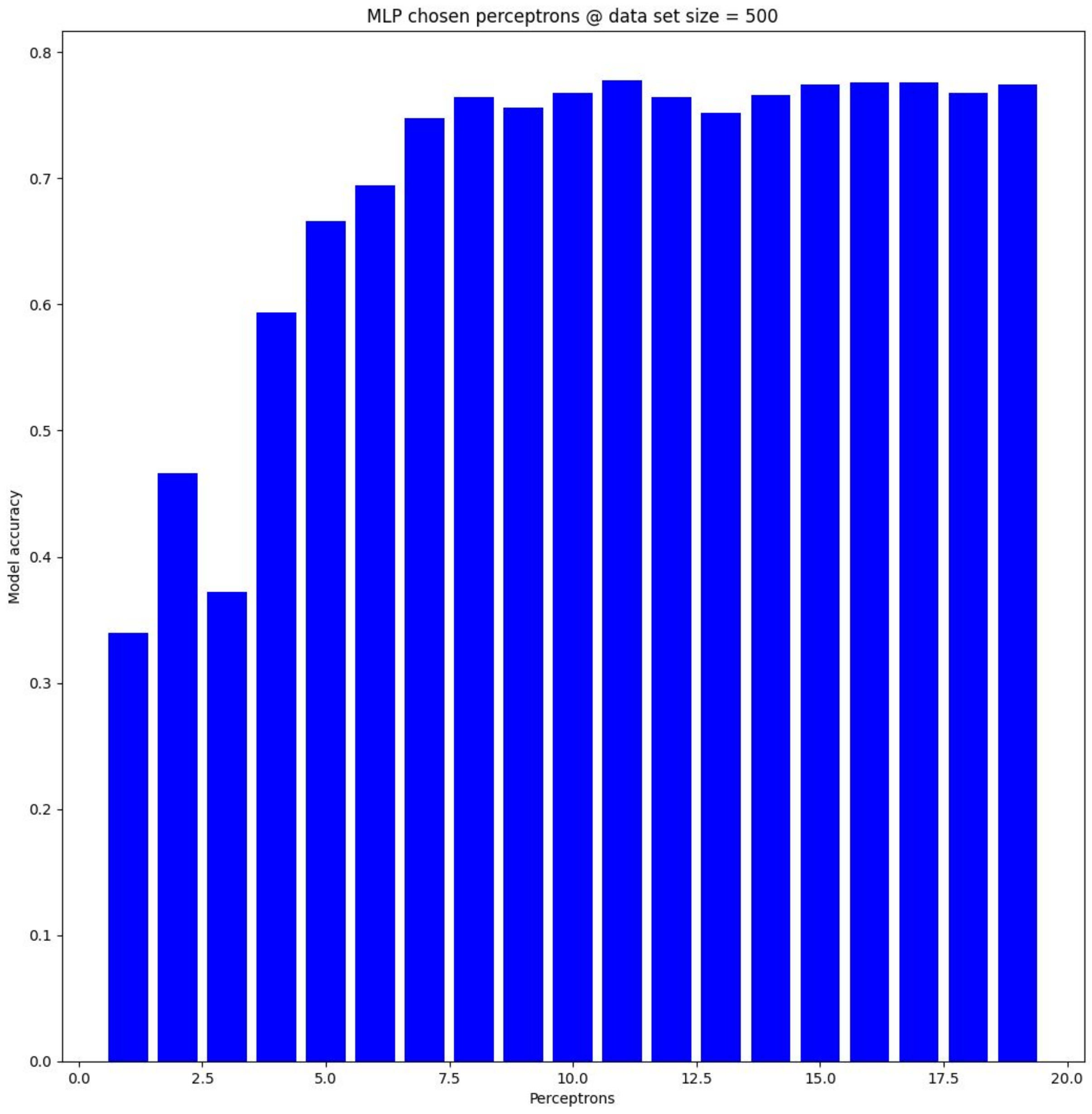
Dataset 100

Percpetron accuracy doesn't really plateau at any particular perceptron value. There is a clear trend though of increased accuracy vs perceptron count. Some perceptrons experience counts result in convergence issues where the minimizer gets stuck at a local minimum at perceptron. This can be seen at perceptron count = 16. Low perceptron counts result in generally bad model performance.



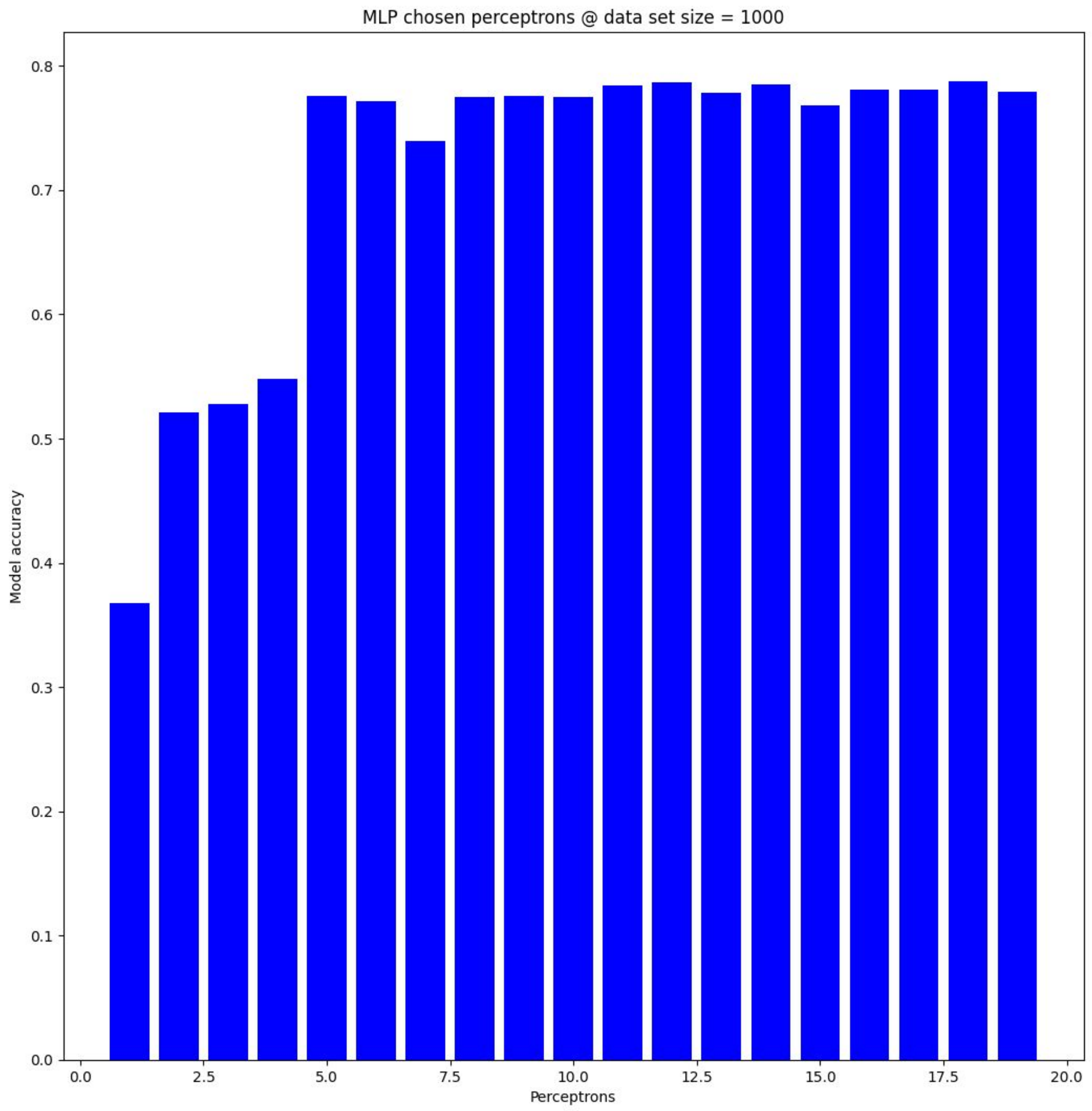
Dataset 500

Percpetron accuracy plateaus at around 8 perceptrons with negligible increase in accuracy at higher perceptron counts. Because all models beyond 8 perceptrons here are very close to the aspirational performance level established by the optimal MAP classifier, the model selected by kfold is only marginally better than the model at the accuracy plateau in the below figure. Additionally, the accuracy from which kfold selects the optimal model is calculated from the mean of the classification accuracy of the models when applied to 1 fold of the training dataset. For small data sizes, that accuracy measurement may not be representative enough of model performance to select the optimal perceptron count.



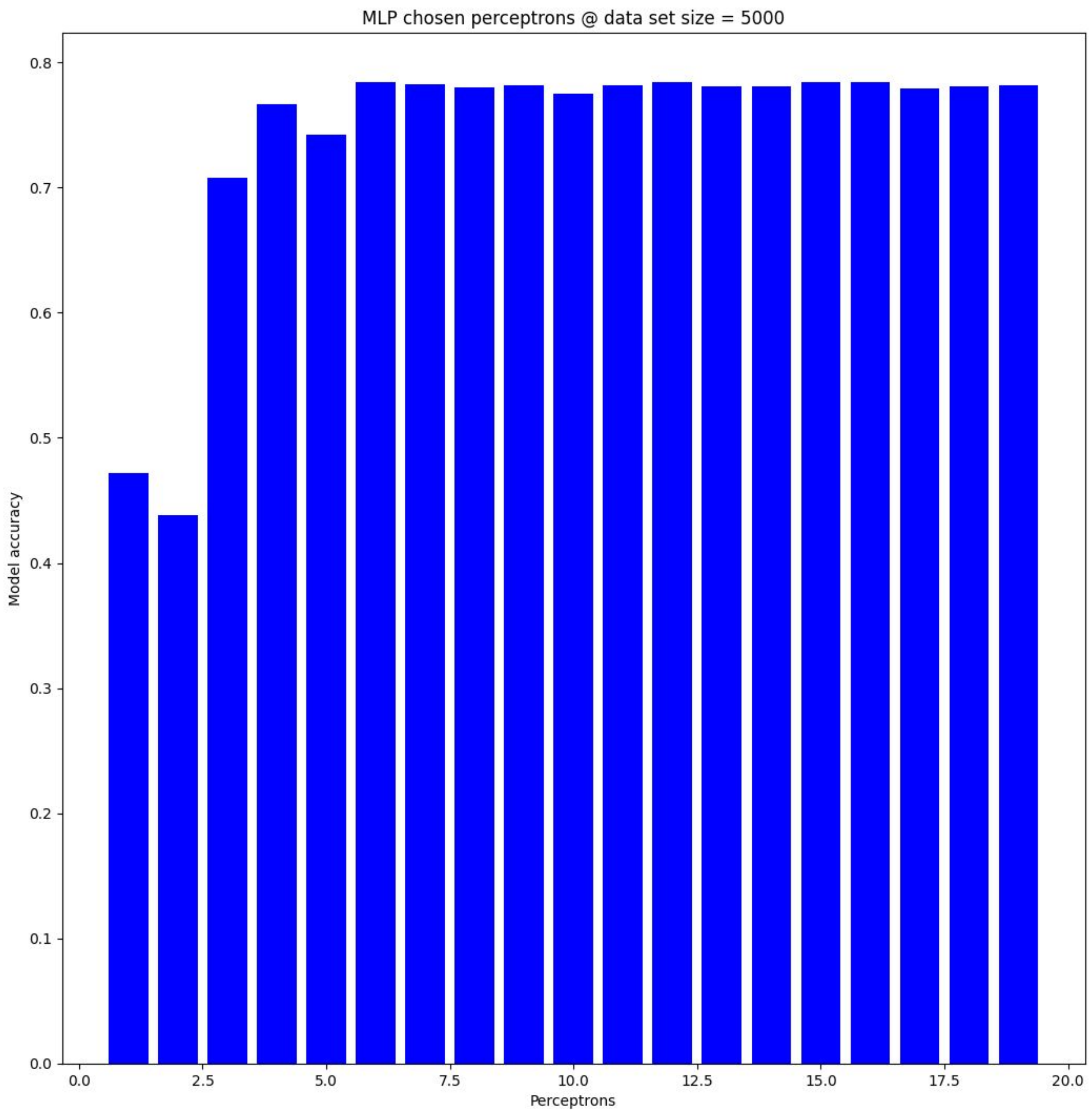
Dataset 1000

Percpetron accuracy plateaus at around 5 perceptrons with negligible increase in accuracy at higher perceptron counts. We are beginning to see a trend here of lower perceptron counts being sufficient for near optimal accuracy performance. More data with fewer perceptrons results in models that can generalize more effectively than more complicated models trained with fewer data that might overfit. Additionally, the 1k dataset with kfold has a larger validation partition to work with when selecting the optimal perceptron count.



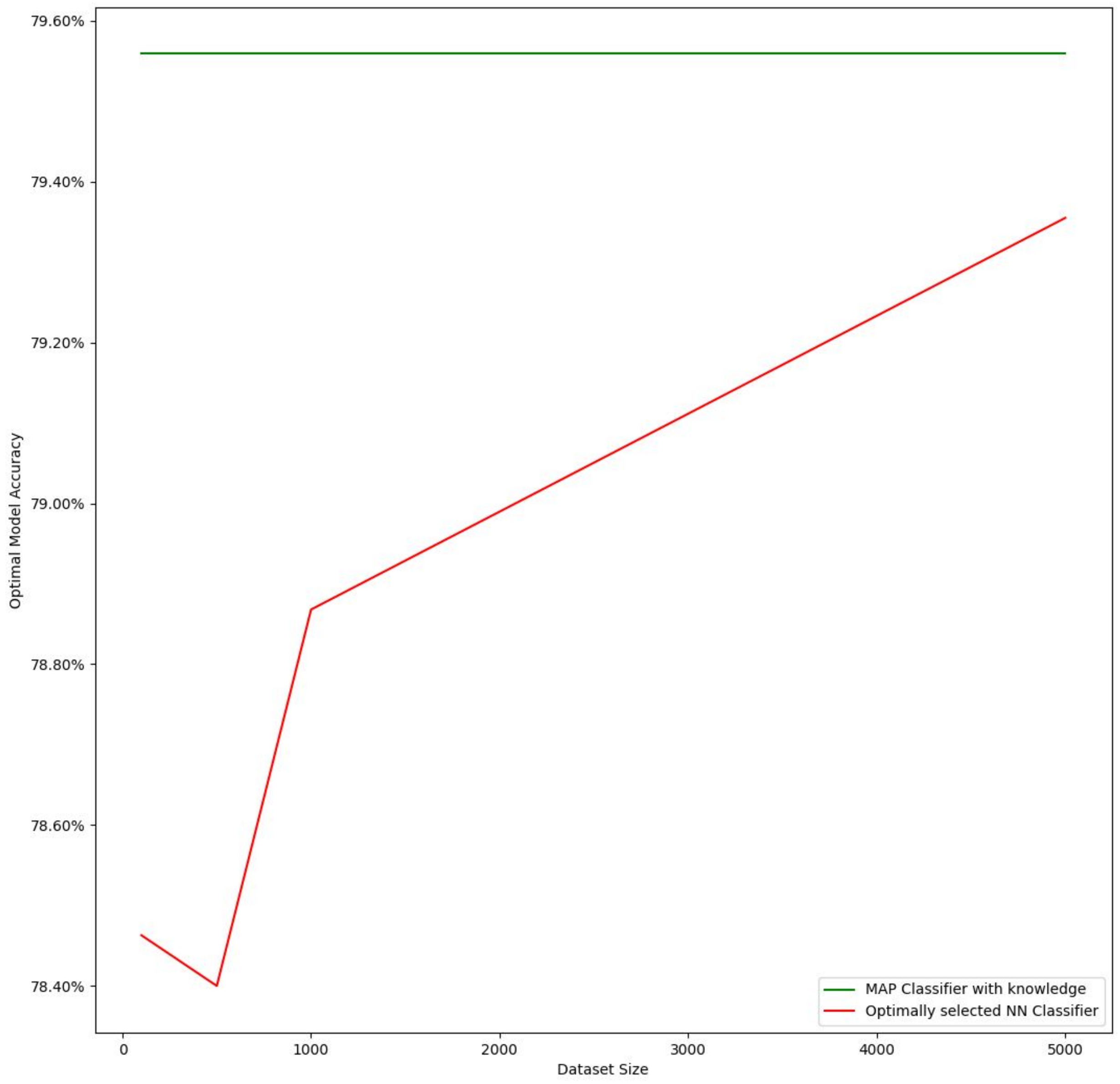
Dataset 5000

Percpetron accuracy plateaus at around 4 perceptrons with negligible increase in accuracy at higher perceptron counts. This observation can be used to establish a trend of MLPs befitting from larger datasets while requiring smaller models to achieve adequate performance. Higher perceptron counts beyond 20 have not been explored but a downwards trend in terms of accuracy is to be expected due to likely overfitting of the model.



Accuracy vs Dataset Size

From the below graph it is clear that more data for MLPs (regardless of model complexity) results in better performing models. Increasing model complexity must be justified by greater data availability. Otherwise overly complex models are unnecessarily computationally expensive and can overfit to the available data.



Code Discussion

A few notes concerning the implementation of Q1:

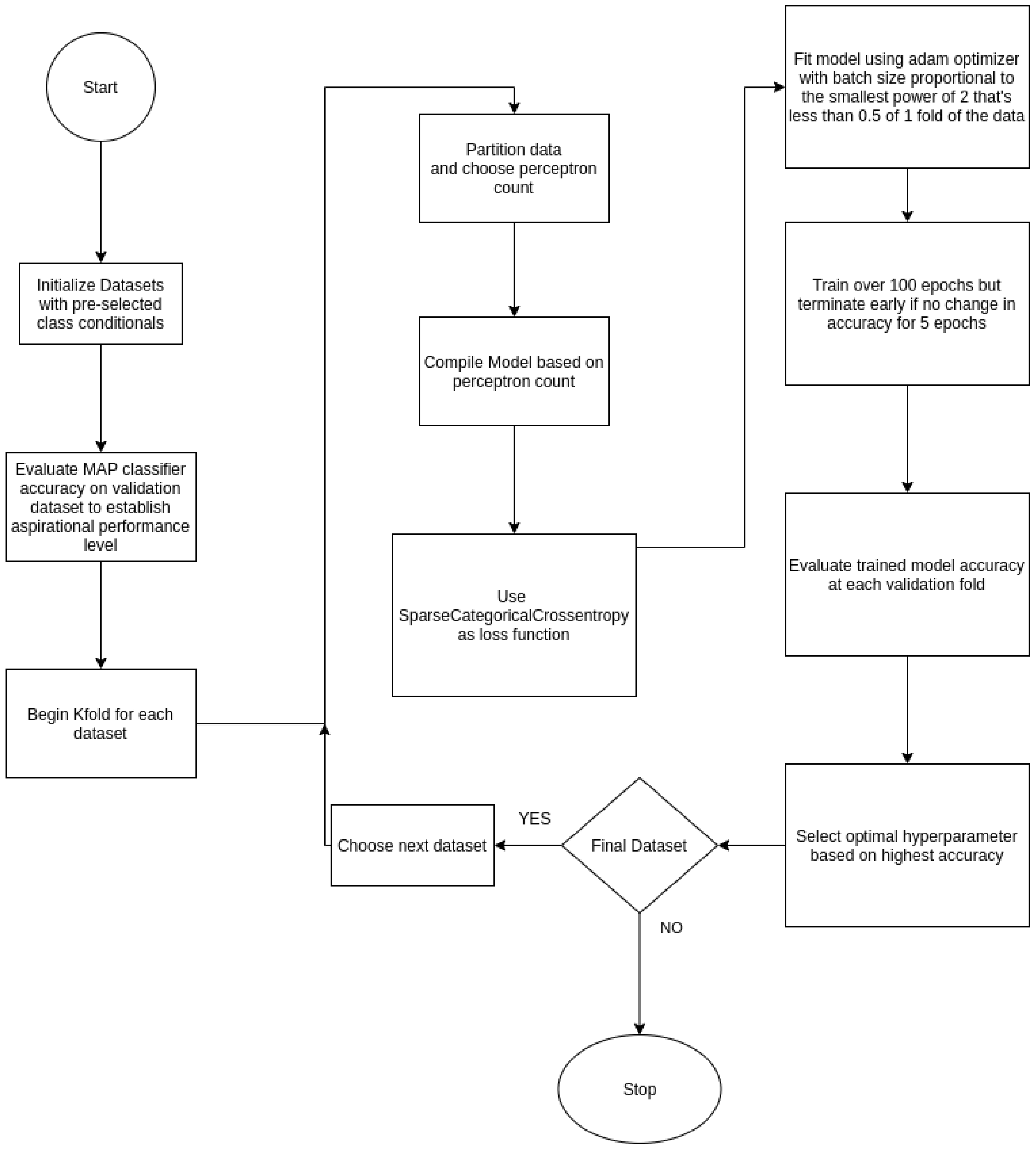
- TensorFlow with Cuda acceleration was used for this question
- SparseCatagoricalLoss was used instead of categorical loss because the label data was not encoded using 1 hot encoding and was instead encoded with integers from 0 to 3
- The models output layer was normalized with softmax so the loss functions from_logits flag was set to False because the outputs of the network are not logits
- Softplus activation was used for the hidden layer as an alternative to Smooth-ReLU
- Training max epochs was set to 100 with early termination using keras callbacks depending on changes in accuracy. If no change in accuracy (subject to a default delta) is observed for 5 epochs the training terminates early and returns the last trained model
- Batch sizes for training were selected based on the following equation:

  2^log(floor((dataset_size/kfolds)*0.50)) i.e. select closes power of 2 that's smaller than ½ the data

  This was done to keep batch sizes to multiples of 2 to speed up communication between CPU and GPU.

Code FlowChart

Below is a general flow chart of the code for this question. Please note that the code was split across two files. Q1.py and analysis.py. Q1 produces the required data for this question and analysis generates the required graphs available above in this report.


Start
Initialize Datasets with pre-selected class conditionals
Evaluate MAP classifier accuracy on validation dataset to establish aspirational performance level
Begin Kfold for each dataset
Partition data and choose perceptron count
Compile Model based on perceptron count
Use SparseCategoricalCrossentropy as loss function
Fit model using adam optimizer with batch size proportional to the smallest power of 2 that's less than 0.5 of 1 fold of the data
Train over 100 epochs but terminate early if no change in accuracy for 5 epochs
Evaluate trained model accuracy at each validation fold
Select optimal hyperparameter based on highest accuracy
Final Dataset
YES
Choose next dataset
NO
Stop

Q2

Chosen Parameters

The parameters: a and **Σ** are randomized everytime the code is run. The below results were generated from the following parameters

a = [56.5067, 65.1923, 23.8135, 72.2253, 41.4401, 62.4820, 77.4701]

Σ =0.5114****I****

Noise variance and L2 Regularizer fluctuation as follows:

$$\in [10^{-3},\ 10^{-3}]$$

$$\in [10^{-3},\ 10^{-2}]$$

MAP parameter estimation derivation

$$y = \omega^T x + \omega_0 + v$$

$$y = \theta^T z + v \qquad v \sim \mathcal{N}(0, 1)$$

$$\omega \in \mathbb{R}^d \qquad \theta = \begin{bmatrix} \omega_0 \\ \omega \end{bmatrix} \quad z = \begin{bmatrix} 1 \\ x \end{bmatrix}$$

prior $\theta \in \mathbb{R}^{d+1}$

$$\theta \sim \mathcal{N}(0, \beta I)$$

$$\hat{\theta}_{map} = \underset{\theta}{\operatorname{argmax}} \ln p(\theta | D)$$

where $D = \{(x_1, y_1), (x_2, y_2), \dots\}$

Assuming iid samples

$$\hat{\theta}_{map} = \underset{\theta}{\operatorname{argmax}} \ln p(D|\theta) + \ln p(\theta)$$

$$= \underset{\theta}{\operatorname{argmax}} \ln p(\theta) + \sum_{i=1}^{N} \ln p(x_i y_i | \omega)$$

$$= \underset{\theta}{\operatorname{argmax}} \ln p(\theta) + \sum_{i=1}^{N} \ln(p(y_i | \omega x_i)\, p(x_i | \omega))$$

Assume $x_i \perp\!\!\!\perp \omega$ $\longrightarrow$ Drop this

$$= \operatorname{argmax} \ln p(\theta) + \sum_{i=1}^{N} \ln(p(y_i | \omega x_i))$$

$$y \sim N(\theta^T z, 1)$$

$$\ln p(\theta) = \ln\left((2\pi)^{\frac{(d+1)}{2}} |\beta I|^{-1/2} e^{-\frac{1}{2}(w-0)^T(\beta I)^{-1}(w-0)}\right)$$

$$= \underbrace{\ln\left((2\pi)^{-d+1/2} |\beta I|^{-1/2}\right)}_{\substack{\text{constant} \\ \text{w.r.t } \theta}} - \frac{1}{2\beta} w^T w$$

$$\ln(p(y|X w) = \ln\left((2\pi)^{-1/2} e^{-\frac{(y-\theta^T z)^2}{2}}\right)$$

$$= \underbrace{-\frac{1}{2}\ln(2\pi)}_{\substack{\text{Constant} \\ \text{w.r.t } \theta}} - \frac{1}{2}(y-\theta^T z)^2$$

$$\hat{\theta}_{MAP} = \underset{\theta}{\arg\max} \ \frac{-1}{2\beta}\theta^T\theta + \sum_{i=1}^{N} \frac{-1}{2}(y_i - \theta^T z_i)^2$$

$$= \underset{\theta}{\arg\min} \ \frac{1}{\beta}\theta^T\theta + \sum_{i=1}^{N}(y_i - \theta^T z_i)^2 \qquad \times(-2)$$

$$\frac{\partial}{\partial \theta^T}\left(\sum_{i=1}^{N}(y_i-\theta^T z_i)^2+\frac{1}{\beta}\theta^T\theta\right)$$

$$=\sum_{i=1}^{N}\frac{\partial}{\partial \theta^T}(y_i-\theta^T z_i)^2+\frac{2\theta}{\beta}$$

$$=-\sum_{i=1}^{N}\cancel{2}(y_i-\theta^T z_i)\,z_i+\frac{\cancel{2}\theta}{\beta}$$

$$=-\sum y_i z_i+\sum \theta^T z_i z_i+\frac{1}{\beta}\theta$$

$$=-\sum y_i z_i+\sum z_i z_i^T\hat{\theta}_{MAP}+\frac{1}{\beta}\hat{\theta}_{MAP}=0$$

$$\sum z_i z_i^T\theta+\frac{1}{\beta}\theta=\sum y_i z_i$$

$$\underbrace{\left(\frac{1}{N}\sum z_i z_i^T+\frac{1}{\beta N}I\right)}_{R(\beta)}\hat{\theta}_{MAP}=\underbrace{\frac{1}{N}\sum y_i z_i}_{q}$$

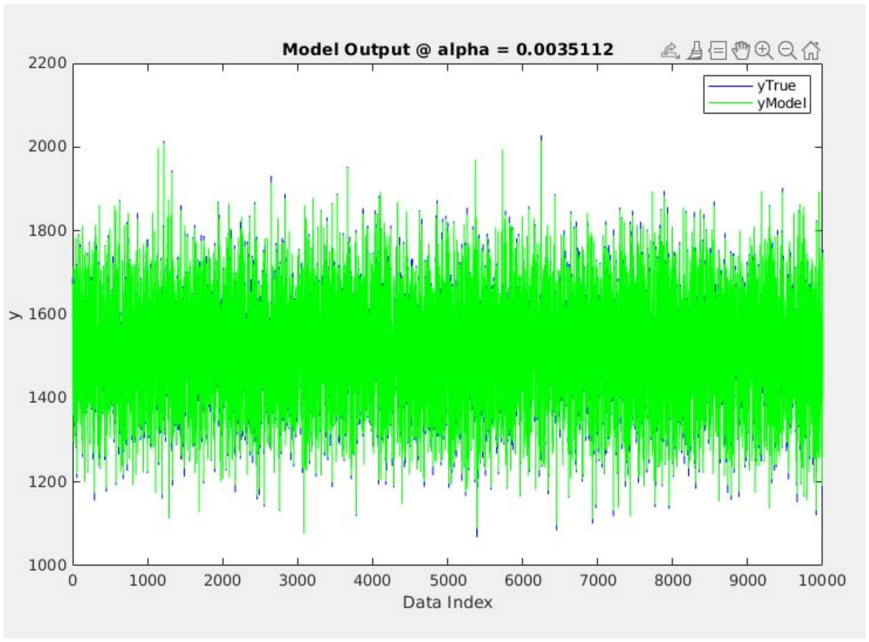
$$\longrightarrow\hat{\theta}_{MAP}=R^{-1}(\beta)\,q\longleftarrow$$

$\lim_{\beta\to\infty}\hat{\theta}_{MAP}=\hat{\theta}_{ML}$ because θ prior becomes irrelevant.

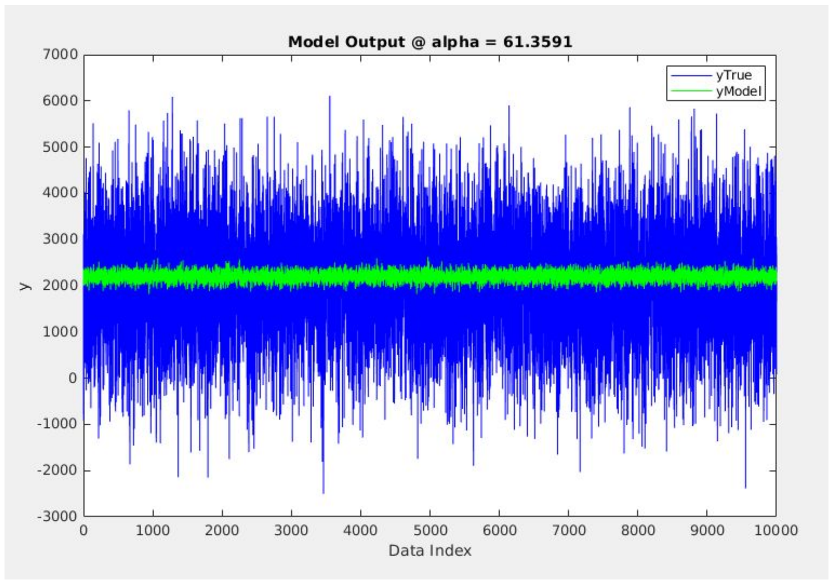
Results

Verifying derivation

At low alpha levels the derived MAP estimator manages to predict the data with high accuracy as shown in the below graph. The achieved MSE for this particular run was 93.4474.



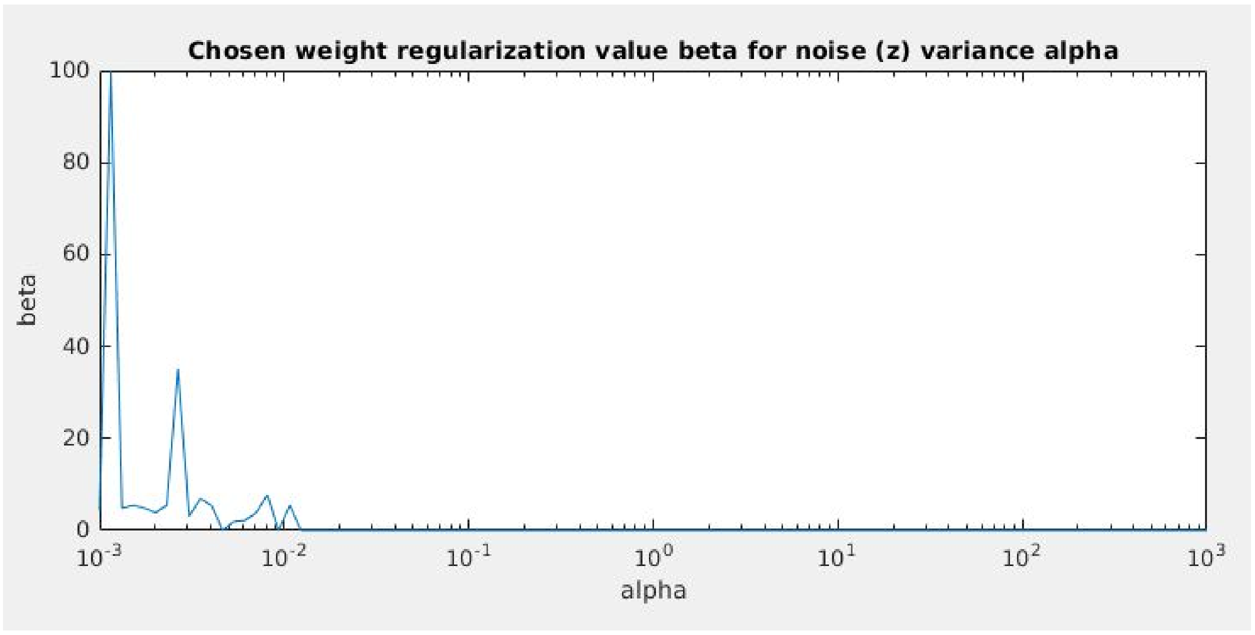
If noise is allowed to dominate the model the the MAP estimator fails to find an appropriate fit. Introducing excess noise affects optimal L2 regularizer selection which will be discussed in the next section.



Results

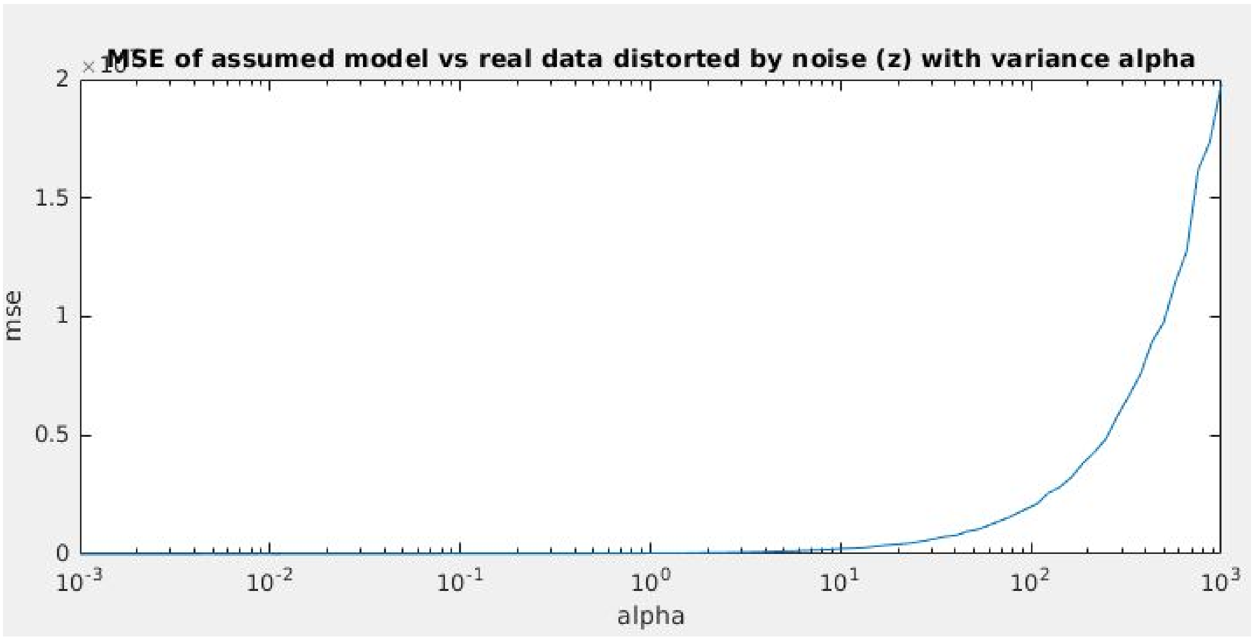
Noise and Regularization

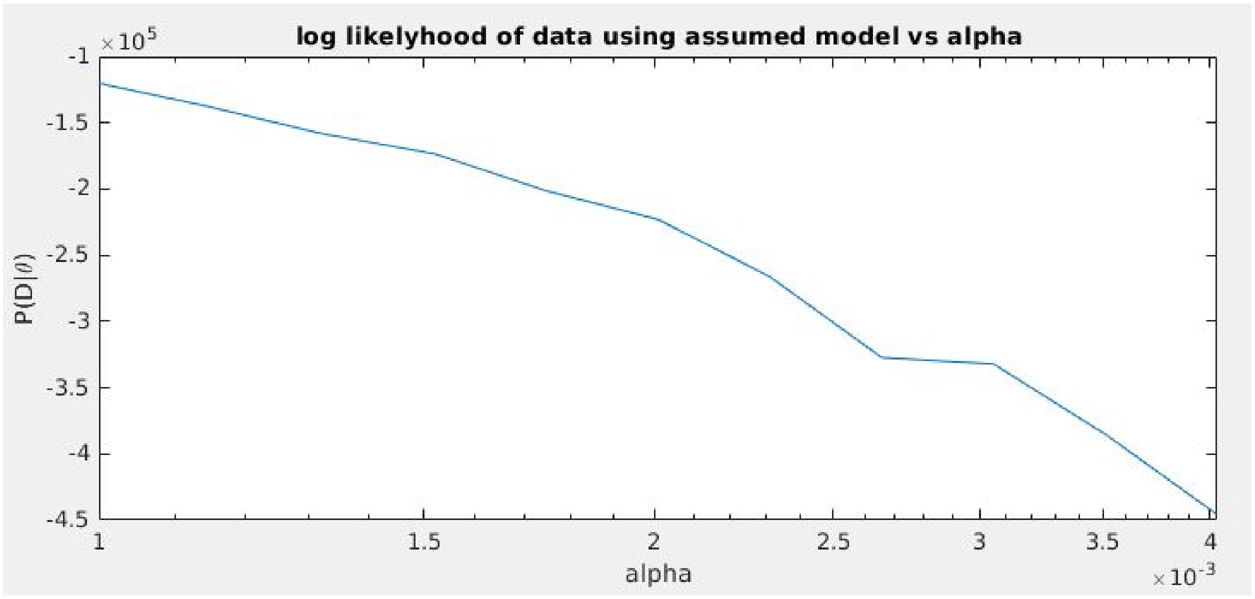
At lower noise levels applied to x, the optimally chosen beta tends towards higher values thus MAP parameter estimation approaches ML parameter estimation which results in the prior on the weights becoming irrelevant. This is understandable because, at low noise levels, the assumed model is close to the actual model of the data. Increasing noise, kfold selects smaller betas to combat the increased variance of the data at high noise levels until eventually it always selects the lowest beta available. At high enough noise levels, noise completely dominates the output of the model rendering regression meaningless.



Data likelihood vs noise

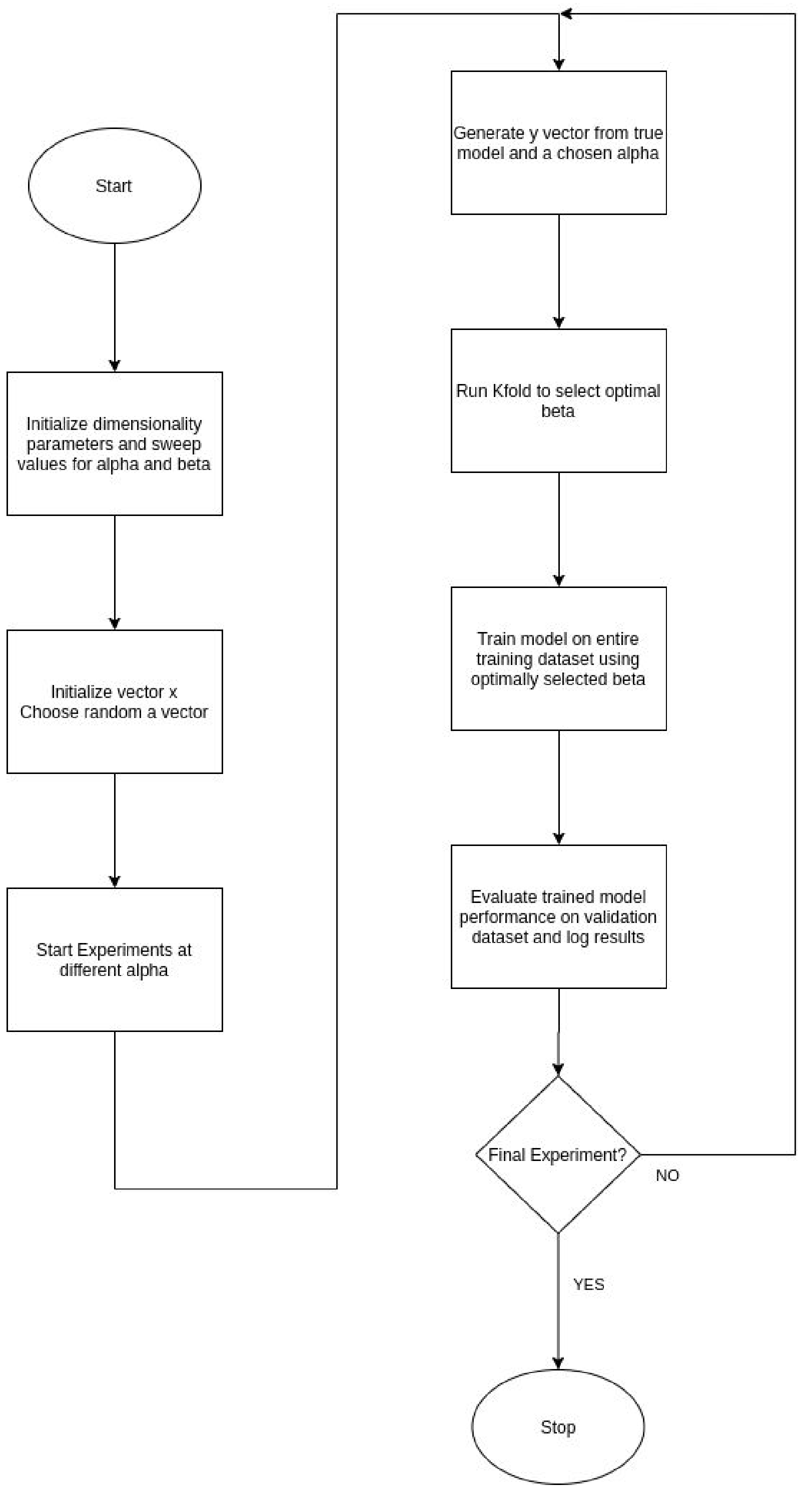
After the optimal beta is selected during kfold, MSE and data likelihood are evaluated on the validation dataset, the below figures show the result of the evaluation at each noise level applied to the data. As expected MSE and data likelihood are heavily affected by the noise introduced in the data. MSE increases sharply at high enough noise variance and data likelihood falls accordingly regardless of the chosen beta.





Code discussion

The overall flow of the code is given in the below figure


Start
Initialize dimensionality parameters and sweep values for alpha and beta
Initialize vector x Choose random a vector
Start Experiments at different alpha
Generate y vector from true model and a chosen alpha
Run Kfold to select optimal beta
Train model on entire training dataset using optimally selected beta
Evaluate trained model performance on validation dataset and log results
Final Experiment?
NO
YES
Stop

Code also available at: https://github.com/asultan123/ML_HW3

Q1 MLP Training Code

```
import os
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.stats import multivariate_normal
import math
import random
from functools import partial
from multiprocessing import Pool
import pickle

class GuassianClassConditionalPdf:
    def __init__(self, mean, sigma):
        self.mean = mean
        self.sigma = sigma
    def generate(self, count):
        return np.random.multivariate_normal(self.mean, self.sigma, count)
    def evaluate(self, data):
        return multivariate_normal(mean=self.mean, cov=self.sigma).pdf(data);

class Data:
    def __init__(self, features = None, labels = None, count = 0, classCount = 0,
gaussianConditionals=None):
        self.features = features
        self.labels = labels
        self.count = count
        self.classCount = classCount
        self.gaussianConditionals = gaussianConditionals
    def generateData(self, gaussianConditionals, count, classPriors = None):
        self.count = count
        self.gaussianConditionals = gaussianConditionals
        self.classCount = len(gaussianConditionals)
        if classPriors is None:
            self.labels = np.array([math.floor(num) for num in
np.random.uniform(0,self.classCount,count)])
            self.labels.sort()
            self.labelCount = [np.count_nonzero(self.labels == l, axis=0) for l in range(0,
self.classCount)]
            self.features = []
            for eachClass in range(0, self.classCount):

self.features.extend(gaussianConditionals[eachClass].generate(self.labelCount[eachClass]).toli
st())
            self.features = np.array(self.features)
            data_aggregate = list(zip(self.labels, self.features))
            random.shuffle(data_aggregate)
            self.labels = np.array([label for label, feature in data_aggregate])
            self.features = np.array([np.array(feature) for label, feature in data_aggregate])
            self.labels = self.labels.astype('float32')
            self.features = self.features.astype('float32')

    def plotData3D(self):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        for l in range(0, self.classCount):
            labelIndexes = np.where(self.labels==l)[0]
            x1 = self.features[labelIndexes,0]
            x2 = self.features[labelIndexes,1]
            x3 = self.features[labelIndexes,2]
            ax.scatter(x1,x2,x3)
        plt.show()
    def mapClassifyWithTrueData(self):
        pxGivenL = []
        for eachClass in range(0, self.classCount):
            pxGivenL.append(self.gaussianConditionals[eachClass].evaluate(self.features))
        classifyResult = np.array(np.argmax(np.array(pxGivenL), axis=0))
        labels = np.array(self.labels)
        accuracy = np.count_nonzero(np.equal(classifyResult, labels))
        return accuracy/len(labels)
    def partition(self, partitionCount):
        elementsPerParition = math.floor(self.count/10)
        paritionedDataFeatures = np.array([self.features[partition:partition+elementsPerParition] for
partition in range(0, self.count, elementsPerParition)])
        paritionedDataLabels = np.array([self.labels[partition:partition+elementsPerParition] for
partition in range(0, self.count, elementsPerParition)])
        dataList = []
        for eachParitionLabels, eachParitionFeatures in zip(paritionedDataLabels,
paritionedDataFeatures):
            dataList.append(Data(eachParitionFeatures, eachParitionLabels, elementsPerParition,
self.classCount, self.gaussianConditionals))
        return dataList
    @classmethod
    def fuse(cls, partitions):
        features = []
        labels = []
        for eachPartition in partitions:
            features.extend(eachPartition.features)
            labels.extend(eachPartition.labels)
        gaussians = partitions[0].gaussianConditionals
        classCount = len(gaussians)
        return Data(np.array(features), np.array(labels), len(features), classCount, gaussians)

class ModelHelper:
    @classmethod
    def createModel(cls, output_size, perceptrons):
        model = tf.keras.models.Sequential([
            tf.keras.layers.Dense(perceptrons, activation='softplus'),
            tf.keras.layers.Dense(output_size, activation='softmax'),
        ])
        loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
        model.compile(optimizer='adam',loss=loss_fn,metrics=['accuracy'])
        return model
    @classmethod
    def fitModel(cls, batch_size, model, data, monitor='accuracy', min_delta=0.001, patience=5,
epochs=100):
        callback = tf.keras.callbacks.EarlyStopping(monitor=monitor, min_delta=min_delta,
patience=patience)
        model.fit(np.array(data.features, dtype=np.float32),np.array(data.labels, dtype=np.float32),
epochs=epochs, batch_size=batch_size, verbose = 0 , callbacks=[callback])
        return model
    @classmethod
    def evalAccuracy(cls, model, data):
        predictions = model(data.features).numpy()
        classifyResult = np.array(np.argmax(predictions, axis=1))
        return (np.count_nonzero(classifyResult==data.labels))/data.count

def kfoldCrossValidation(kfolds, dataset, modelCreator, modelFitter, modelEvaluater,
hyperparamater_range):
    def keywithmaxval(d):
        v=list(d.values())
        k=list(d.keys())
        return k[v.index(max(v))]

    hyperparameter_accuracy = {}
    for hyperparameter in hyperparamater_range:
        print('Hyperparameter = ' + str(hyperparameter))
        model = modelCreator(hyperparameter)
        partitionedDataset = dataset.partition(kfolds)
        accuracy = [0]*kfolds
        for k in range(0, kfolds):
            print('K = ' + str(k))
            trainingData = Data.fuse(partitionedDataset[0:k] +
partitionedDataset[k+1:len(partitionedDataset)])
            validationData = partitionedDataset[k:k+1][0]
            fittedModel = modelFitter(model, trainingData)
            accuracy[k] = modelEvaluater(model, validationData)
        hyperparameter_accuracy[hyperparameter] = np.mean(accuracy)

    optimal_hyperparameter = keywithmaxval(hyperparameter_accuracy)
    optimal_model = modelCreator(optimal_hyperparameter)
    fitted_optimal_model = modelFitter(optimal_model, dataset)

    return (optimal_hyperparameter, hyperparameter_accuracy, fitted_optimal_model)


if __name__ == "__main__":
    print(tf.test.is_gpu_available())
    d1 = Data()
    d2 = Data()
    d3 = Data()
    d4 = Data()
    dvalid = Data()
    c1 = GuassianClassConditionalPdf([1,0,0], [[1,0,0],[0,1,0],[0,0,1]])
    c2 = GuassianClassConditionalPdf([0,0,2.5], [[1,0,0],[0,1,0],[0,0,1]])
    c3 = GuassianClassConditionalPdf([0,0.5,0], [[1,0,0],[0,1,0],[0,0,1]])
    c4 = GuassianClassConditionalPdf([1,0,10], [[1,0,0],[0,1,0],[0,0,1]])
    d1.generateData([c1, c2, c3, c4], 100)
    d2.generateData([c1, c2, c3, c4], 500)
    d3.generateData([c1, c2, c3, c4], 1000)
    d4.generateData([c1, c2, c3, c4], 5000)
    datasets = [d1, d2, d3, d4]
    dvalid.generateData([c1, c2, c3, c4], 100000)

    kfolds = 10
    results = {}
    for eachDataset in datasets:
        print('Dataset: ' + str(eachDataset.count))

        results[str(eachDataset.count)] = {}
        results[str(eachDataset.count)]['map_accuracy'] =
eachDataset.mapClassifyWithTrueData()
        results[str(eachDataset.count)]['dataset'] = {}
        results[str(eachDataset.count)]['optimal_hyperparameter'] = {}
        results[str(eachDataset.count)]['hyperParameter_accuracy'] = {}
        results[str(eachDataset.count)]['fittedModel_validation_accuracy'] = {}

        modelCreator = partial(ModelHelper.createModel, 4)
        batch_size = 2**int(math.log((eachDataset.count/kfolds)*0.50,2))
        print('Dataset batch size: ' + str(batch_size))

        modelFitter = partial(ModelHelper.fitModel, batch_size)

        (optimal_hyperparameter, hyperParameter_accuracy, fitted_optimal_model) =
kfoldCrossValidation(kfolds, eachDataset, modelCreator, modelFitter,
ModelHelper.evalAccuracy, hyperparamater_range = range(1,5))
        fittedModel_validation_accuracy = ModelHelper.evalAccuracy(fitted_optimal_model, dvalid)

        results[str(eachDataset.count)]['dataset'] = eachDataset
        results[str(eachDataset.count)]['optimal_hyperparameter'] = optimal_hyperparameter
        results[str(eachDataset.count)]['hyperParameter_accuracy'] = hyperParameter_accuracy
        results[str(eachDataset.count)]['fittedModel_validation_accuracy'] =
fittedModel_validation_accuracy
        with open('TEST_final_results.pkl', 'wb') as output:
            pickle.dump(results, output, pickle.HIGHEST_PROTOCOL)

    print('Done')
```

Q1 Analysis Code

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import multivariate_normal
import math
import random
from functools import partial
from multiprocessing import Pool
import pickle
import pprint


class GuassianClassConditionalPdf:
    def __init__(self, mean, sigma):
        self.mean = mean
        self.sigma = sigma
    def generate(self, count):
        return np.random.multivariate_normal(self.mean, self.sigma, count)
    def evaluate(self, data):
        return multivariate_normal(mean=self.mean, cov=self.sigma).pdf(data);

class Data:
    def __init__(self, features = None, labels = None, count = 0, classCount = 0,
gaussianConditionals=None):
        self.features = features
        self.labels = labels
        self.count = count
        self.classCount = classCount
        self.gaussianConditionals = gaussianConditionals
    def generateData(self, gaussianConditionals, count, classPriors = None):
        self.count = count
        self.gaussianConditionals = gaussianConditionals
        self.classCount = len(gaussianConditionals)
        if classPriors is None:
            self.labels = np.array([math.floor(num) for num in
np.random.uniform(0,self.classCount,count)])
```

```
            self.labels.sort()
            self.labelCount = [np.count_nonzero(self.labels == l, axis=0) for l in range(0,
self.classCount)]
            self.features = []
            for eachClass in range(0, self.classCount):

self.features.extend(gaussianConditionals[eachClass].generate(self.labelCount[eachClass]).toli
st())
            self.features = np.array(self.features)
            data_aggregate = list(zip(self.labels, self.features))
            random.shuffle(data_aggregate)
            self.labels = np.array([label for label, feature in data_aggregate])
            self.features = np.array([np.array(feature) for label, feature in data_aggregate])
            self.labels = self.labels.astype('float32')
            self.features = self.features.astype('float32')

    def plotData3D(self):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        for l in range(0, self.classCount):
            labelIndexes = np.where(self.labels==l)[0]
            x1 = self.features[labelIndexes,0]
            x2 = self.features[labelIndexes,1]
            x3 = self.features[labelIndexes,2]
            ax.scatter(x1,x2,x3)

        ax.set_xlabel('X1 Label')
        ax.set_ylabel('X2 Label')
        ax.set_zlabel('X3 Label')
        ax.set_title('Data distribution')
        plt.show()
        return fig

    def mapClassifyWithTrueData(self):
        pxGivenL = []
        for eachClass in range(0, self.classCount):
            pxGivenL.append(self.gaussianConditionals[eachClass].evaluate(self.features))
        classifyResult = np.array(np.argmax(np.array(pxGivenL), axis=0))
        labels = np.array(self.labels)
        accuracy = np.count_nonzero(np.equal(classifyResult, labels))
        return accuracy/len(labels)
    def partition(self, partitionCount):
        elementsPerParition = math.floor(self.count/10)
        paritionedDataFeatures = np.array([self.features[partition:partition+elementsPerParition] for
partition in range(0, self.count, elementsPerParition)])
        paritionedDataLabels = np.array([self.labels[partition:partition+elementsPerParition] for
partition in range(0, self.count, elementsPerParition)])
        dataList = []
        for eachParitionLabels, eachParitionFeatures in zip(paritionedDataLabels,
paritionedDataFeatures):
            dataList.append(Data(eachParitionFeatures, eachParitionLabels, elementsPerParition,
self.classCount, self.gaussianConditionals))
        return dataList
    @classmethod
    def fuse(cls, partitions):
        features = []
        labels = []
        for eachPartition in partitions:
            features.extend(eachPartition.features)
            labels.extend(eachPartition.labels)
        gaussians = partitions[0].gaussianConditionals
        classCount = len(gaussians)
        return Data(np.array(features), np.array(labels), len(features), classCount, gaussians)

class ModelHelper:
    @classmethod
    def createModel(cls, output_size, perceptrons):
        model = tf.keras.models.Sequential([
            tf.keras.layers.Dense(perceptrons, activation='softplus'),
            tf.keras.layers.Dense(output_size, activation='softmax'),
        ])
        loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
        model.compile(optimizer='adam',loss=loss_fn,metrics=['accuracy'])
        return model
    @classmethod
    def fitModel(cls, batch_size, model, data, monitor='accuracy', min_delta=0.001, patience=5,
epochs=100):
        callback = tf.keras.callbacks.EarlyStopping(monitor=monitor, min_delta=min_delta,
patience=patience)
        model.fit(np.array(data.features, dtype=np.float32),np.array(data.labels, dtype=np.float32),
epochs=epochs, batch_size=batch_size, verbose = 0 , callbacks=[callback])
        return model
    @classmethod
    def evalAccuracy(cls, model, data):
        predictions = model(data.features).numpy()
        classifyResult = np.array(np.argmax(predictions, axis=1))
        return (np.count_nonzero(classifyResult==data.labels))/data.count
```

```
if __name__ == "__main__":
    with open('final_results.pkl', 'rb') as input:
        results = pickle.load(input)
        pp = pprint.PrettyPrinter(indent=4)
        pp.pprint(results)

    # plot 5k dataset
    fig = results['5000']['dataset'].plotData3D()

    # plot histograms of perceptron accuracies
    for dataset_key, dataset_results in results.items():
        fig = plt.figure()
        plt.bar(range(1,20), list(dataset_results['hyperParameter_accuracy'].values()), color='blue')
        plt.xlabel('Perceptrons')
        plt.ylabel('Model accuracy')
        plt.title('MLP chosen perceptrons @ data set size = ' + dataset_key)
        plt.show()

    # Plot optimal perceptrons
    chosenPerceptrons = []
    datasetSizes = []
    for dataset_key, dataset_results in results.items():
        chosenPerceptrons.append(dataset_results['optimal_hyperparameter'])
        datasetSizes.append(dataset_results['dataset'].count)
    print(datasetSizes)
    print(chosenPerceptrons)
    plt.bar(datasetSizes, chosenPerceptrons, width=100, color='b')
    plt.xlabel('Dataset Sizes')
    plt.ylabel('Optimal Perceptron Count')
    plt.xticks(datasetSizes)
    plt.yticks(chosenPerceptrons)
    plt.show()

    # Map Accuracy vs Optimal NN accuracy
    map_accuracy = []
    optimal_NN_accuracy = []
    datasetSizes = []
    gaussians = results['5000']['dataset'].gaussianConditionals
    dValid = Data()
    dValid.generateData(gaussians,100000)
    for dataset_key, dataset_results in results.items():
        map_accuracy.append(dValid.mapClassifyWithTrueData())
        datasetSizes.append(dataset_results['dataset'].count)
        optimal_NN_accuracy.append(dataset_results['fittedModel_validation_accuracy'])
    plt.plot(datasetSizes,map_accuracy, color='g', label='MAP Classifier with knowledge')
    plt.plot(datasetSizes,optimal_NN_accuracy, color='r', label='Optimally selected NN Classifier')
    plt.xlabel('Dataset Size')
    plt.ylabel('Optimal Model Accuracy')
    plt.gca().set_yticklabels(['{:.2f}%'.format(x*100) for x in plt.gca().get_yticks()])
    plt.legend()
    plt.show()
```

Q2 Code

```
clear all, close all

% ======================== Parameter Setup ==========================%

n=7;
Ntrain=100;
Ntest=10000;
lower_noise_multiplier = 10^-3;
upper_noise_multiplier = 10^3;
x_mu_max=10;
totalExp = 100;
betasPerExperiment = 100;
maxExpPerJob = 10;
maxJobs = floor(totalExp/maxExpPerJob);
betaMin = -3;
betaMax = 2;

% ======================== Data Generation ==========================%

[x, gm] = generateRandGMMData(1, n, Ntrain+Ntest, x_mu_max);
wtrue = rand(n,1)*100;

% alpha = trace(gm.Sigma(:,:,1))/n*10.^(linspace(-2,2, totalExp));
alpha = logspace(-3,3, totalExp);
expRange = floor(linspace(0,totalExp-1,maxExpPerJob));

chosenBetas = zeros(1,totalExp);
mse = zeros(1,totalExp);
data_likelyhood = zeros(1,totalExp);
```

```
% ============= Experiment Runs at Increasing Alphas ===============%

for exp = 1:totalExp

exp,

% ============= Generate Noisy Data ===============%

[y, x] = noisy_model(n, Ntrain+Ntest, wtrue, x, alpha(exp));
data = segment_data(y, x, Ntrain);

% ============= Use Kfold to optimize L2 Regularizer for Wmap ===============%

[bestBeta, wfit] = runkfoldexperiment_ridgeregression_bestbeta(Ntrain, 10, betaMin, betaMax,
betasPerExperiment, data.train, 1);


experiment{exp}.data = data;
chosenBetas(exp) = bestBeta;
chosenWfits{exp} = wfit;

% ============= Evaluate MSE for optimally L2 Regularizer ===============%

[~, total_mse, ~] = calc_MSE_noisy_linear_model(data.test.y,data.test.x,wfit);
mse(exp) = total_mse;

data_likelyhood(exp) = data_log_likelyhood(data.test.y,data.test.x, wfit, 1);
end

figure(1)
subplot(3,1,1), semilogx(alpha, chosenBetas)
title('Chosen weight regularization value beta for noise (z) variance alpha');
xlabel('alpha'),ylabel('beta'),
hold on, subplot(3,1,2), semilogx(alpha, mse)
title('MSE of assumed model vs real data distorted by noise (z) with variance alpha');
xlabel('alpha'),ylabel('mse'),
hold on, subplot(3,1,3), semilogx(alpha, data_likelyhood)
title('log likelyhood of data using assumed model vs alpha');
xlabel('alpha'),ylabel('P(D|{\theta})'),

figure(2),
subplot(2,1,1),
plot_experiment_data(10, alpha, experiment, chosenWfits)
title(['Model Output @ alpha = ', num2str(alpha(10))]);
```

```
xlabel('Data Index'),ylabel('y'), legend('yTrue', 'yModel');

subplot(2,1,2),
plot_experiment_data(80, alpha, experiment, chosenWfits)
title(['Model Output @ alpha = ', num2str(alpha(80))]);
xlabel('Data Index'),ylabel('y'), legend('yTrue', 'yModel');

function plot_experiment_data(exp, alpha, experiment, chosenWfits)
noise_alpha = alpha(exp);
noise_alpha
[pred, total_mse, ~] =
calc_MSE_noisy_linear_model(experiment{exp}.data.test.y,experiment{exp}.data.test.x,chosen
Wfits{exp});
total_mse
plot_data_and_noise_model_pred(experiment{exp}.data.test.y, pred)
end


function [pred, total_mse, mse] = calc_MSE_noisy_linear_model(y,x,wfit)
   N = size(x,2);
   b = [ones(1,N); x];
   pred = wfit'*b;
   mse = (y-wfit'*b).^2;
   total_mse = mean(mse);
end

function [bestBeta, wfit] = runkfoldexperiment_ridgeregression_bestbeta(N, K, betaMin,
betaMax, betaCount, dataset, model_noise_variance)
   partitions_idx_start = ceil(linspace(0,N,K+1));
   indPartitionLimits = zeros(K, 2);
   for k = 1:K
      indPartitionLimits(k,:) = [partitions_idx_start(k)+1,partitions_idx_start(k+1)];
   end
   loglikelyood_Beta = zeros(1,betaCount);
   beta = 10.^linspace(betaMin, betaMax, betaCount);
   for betaIdx = 1:betaCount
      loglikelyhood = zeros(1,K);
      for k= 1:K
         indValidate = indPartitionLimits(k,1):indPartitionLimits(k,2);
         x = dataset.x;
         y = dataset.y;
         xValidate = x(:, indValidate); % Using folk k as validation set
         yValidate = y(:, indValidate);
         if k == 1
```

```
            indTrain = indPartitionLimits(k+1,1):N;
        elseif k == K
            indTrain = 1:indPartitionLimits(k-1,2);
        else
            indTrain = [1:indPartitionLimits(k-1,2),indPartitionLimits(k+1,1):N];
        end
        xTrain = x(:, indTrain); % using all other folds as training set
        yTrain = y(:, indTrain); % using all other folds as training set

        wfit = fit_noisy_linear_model(yTrain, xTrain, beta(betaIdx), model_noise_variance);
%           [~, total_mse, ~] = calc_MSE_noisy_linear_model(yValidate,xValidate,wfit);
%           loglikelyhood(k) = total_mse;

        loglikelyhood(k) = data_log_likelyhood(yValidate,xValidate, wfit, model_noise_variance);
      end
      loglikelyood_Beta(betaIdx) = mean(loglikelyhood);
   end
   [~, bestBetaIdx] = max(loglikelyood_Beta);
%     [~, bestBetaIdx] = min(loglikelyood_Beta);
   bestBeta = beta(bestBetaIdx);
   wfit = fit_noisy_linear_model(dataset.y, dataset.x, bestBeta, model_noise_variance);
end

function plot_data_and_noise_model_pred(y, pred)
   plot(y,'b'), hold on
   plot(pred, 'g'),
end

function data = segment_data(y, x, Ntrain)
   data.train.x = x(:, 1:Ntrain);
   data.train.y = y(:, 1:Ntrain);
   data.test.x = x(:, Ntrain+1:end);
   data.test.y = y(:, Ntrain+1:end);
end

function [y, x] = noisy_model(n, N, wtrue, x, alpha)
z = generateRandWhiteNoise(1, n, N, alpha);
v = generateRandWhiteNoise(1, 1, N, 1);
y = wtrue'*(x+z)+v;
end

function log_likelyhood = data_log_likelyhood(y, x, w, noise_variance)
   N = size(x,2);
   b = [ones(1,N); x];
```

```
%     log_likelyhood =
sum(log((1/(sqrt(2*pi)*noise_variance))*exp(-(y-w'*b)/(2*(noise_variance)^2))));
   likelyhoods = zeros(1,N);
   mus = w'*b;
   for i = 1:N
      likelyhoods(i) = evalGaussian(y(i), mus(i), noise_variance.^2);
   end
   log_likelyhood = sum(log(likelyhoods));
end

function g = evalGaussian(x,mu,Sigma)
% Evaluates the Gaussian pdf N(mu,Sigma) at each coumn of X
   [n,N] = size(x);
   C = ((2*pi)^n * det(Sigma))^(-1/2);
   E = -0.5*sum((x-repmat(mu,1,N)).*(inv(Sigma)*(x-repmat(mu,1,N))),1);
   g = C*exp(E);
end

function wfit = fit_noisy_linear_model(y,x, beta, noise_variance)
   N = size(x,2);
   dim = size(x,1)+1;
   b = [ones(1,N); x];
   y = repmat(y, dim, 1);
   R = b*b';
   Q = sum(y.*b, 2);
   wfit = inv(R+(noise_variance^2/beta).*eye(dim))*Q;
end

function [data, gmdist] = generateRandWhiteNoise(Order, dim, N, alpha)
   sigma = zeros(dim,dim,Order);
   mu = zeros(Order, dim);
   for m = 1:Order
      sigma(:,:,m) = alpha.*eye(dim);
   end
   gmdist = gmdistribution(mu,sigma,ones(1,Order)/Order);
   data = random(gmdist,N)';
end

function [data, gmdist] = generateRandGMMData(Order, dim, N, mu_max)
   sigma = zeros(dim,dim,Order);
   mu = zeros(Order, dim);
   for m = 1:Order
      sigma(:,:,m) = rand(1,1).*eye(dim);
      mu(m, :) = rand(1,dim)*mu_max;
```

```
    end
    gmdist = gmdistribution(mu,sigma,ones(1,Order)/Order);
    data = random(gmdist,N)';
end
```