



Pattern Recognition Assignment

(2023 - 2024)

Team 42

Kyparissis Kyparissis
Fotios Alexandridis

10346
9953


kyparkypar@ece.auth.gr
faalexandr@ece.auth.gr



Introduction

Part A, B, C

- Experiment with different classification models
- Plot decision boundaries and get metrics' results
- Compare models using the same training subset

Models are mostly created and trained using the  scikit-learn python library

- Provides many ready-to-go functions for
 - ML model creation
 - Dataset preprocessing
 - Metrics
 - Visualization

For comparing the models, we use the mean classification error metric (Zero-One Classification Loss)

$$e = \frac{\sum \text{Wrongly classified test samples}}{\sum \text{Test samples}} = (1 - \text{accuracy})$$

We will work on the dataset . csv dataset (provided by the assignment)

- We used GitHub and Git to import the dataset into our  Google Colab notebook

Dataset

Import, Exploration/Visualization and Preprocessing

(1)

dataset.csv

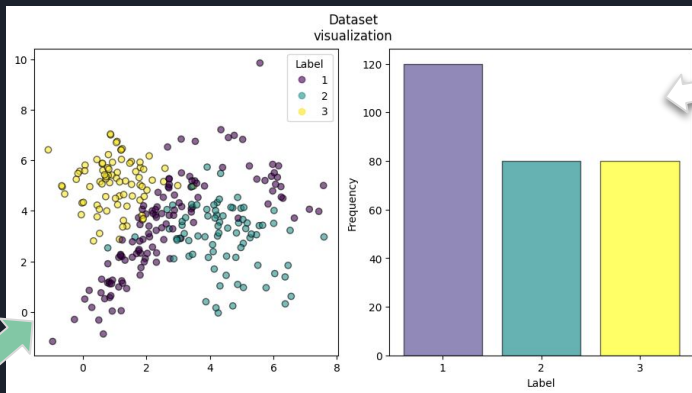
	X		y
	(Feature #1)	(Feature #2)	Label
0	1.80360	4.4229	3
1	3.46150	4.1436	2
2	2.18730	3.9964	1
3	3.09330	2.9056	1
4	1.75860	2.4109	1
...
275	-0.94428	-1.1722	1
276	1.21990	6.0341	3
277	0.91094	1.1120	1
278	6.24770	4.5430	1

280 rows × 3 columns

Small 2D
Dataset

	(Feature #1)	(Feature #2)
count	280.000000	280.000000
mean	2.743918	3.766240
std	1.908206	1.761147

Almost normalized data



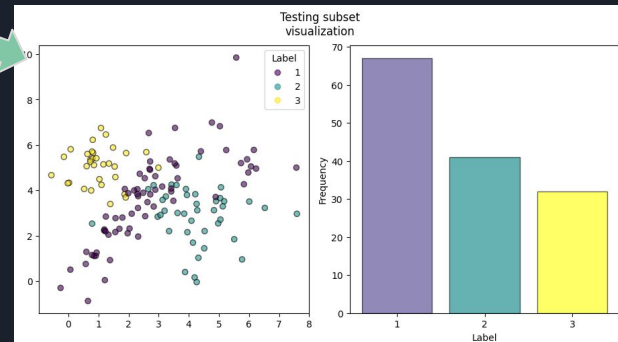
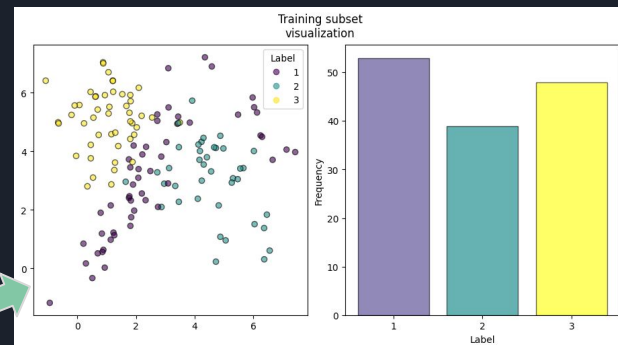
Split Dataset

- 50% Training
- 50% Testing

```
1 split_scale = 0.5 # Split into training and test sets with a 50% - 50% ratio
2
3 X_train, X_test, y_train, y_test = train_test_split(X, y,
4                                                    test_size=split_scale,
5                                                    shuffle=True,
6                                                    random_state=42)
```

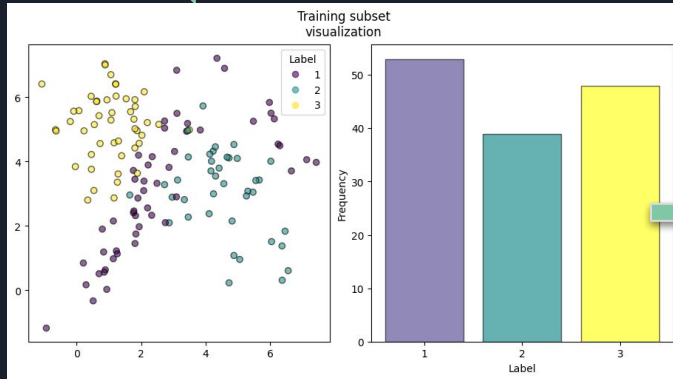
We see that we have:

- 3 Classes
- No outliers
- Imbalanced (labels) dataset

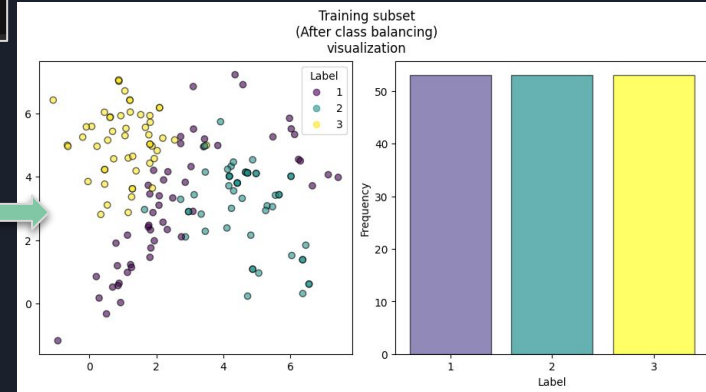
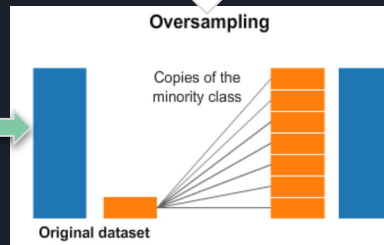


Dataset

Import, Exploration/Visualization and Preprocessing (2) (Class Balancing)



```
1 ros = RandomOverSampler(random_state=42)
2 X_train, y_train = ros.fit_resample(X_train, y_train)
```



- Training dataset is **NOT** balanced
 - All labels are not on the same (or almost) level / height
- This can create problems during the training and testing process
 - Biases

We fix this issue using Oversampling / Undersampling techniques
We choose to **Oversample** the dataset because:

- Small Dataset
 - Only 140 samples in the training dataset
 - Undersampling would shrink the dataset even more
 - Not enough data for training
 - Overfitting and lack of generalization

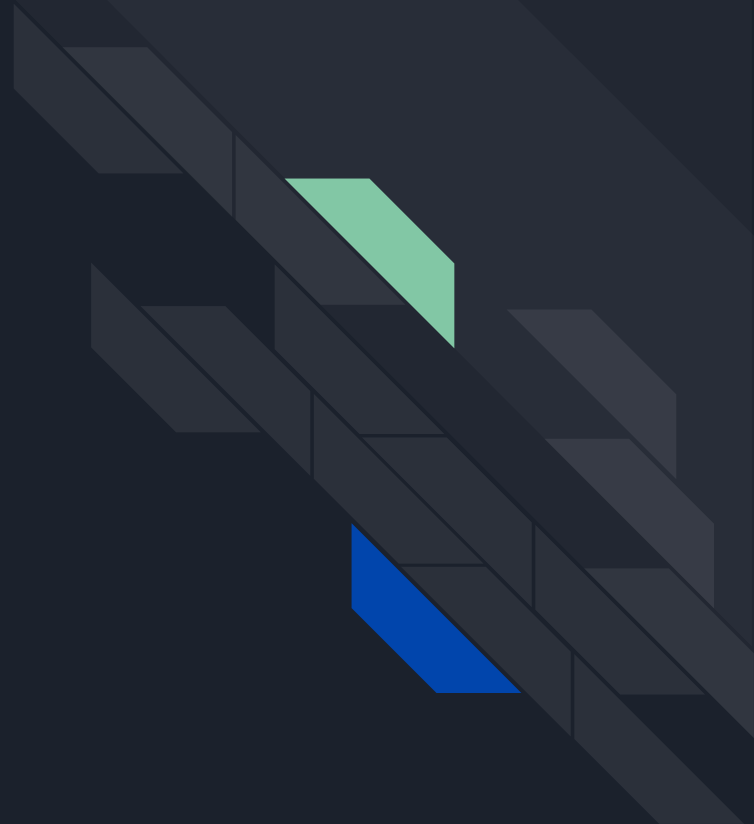
We keep the testing dataset as is so we can have a realistic indication of the performance of our models!

Part A

Bayes Classifier

Create a Bayes classifier that uses:

- Maximum Likelihood method for parameter estimation
- Assuming normal distributed data
- Assuming:
 - Same Covariance Matrix for all classes
 - Different Covariance Matrix for each class



Part A

Bayes Classifier

- In theory, we would always like to predict a qualitative response with the Bayes classifier
 - Gives us the lowest test error rate out of all classifiers

BUT

- We need to know the true conditional population distribution of Y given X
 - We do not know if X is truly drawn from a normal distribution
- We have to know the true population parameters



Approximation
using

LDA
(Linear Discriminant Analysis)

QDA
(Quadratic Discriminant Analysis)

, since according to Bayes rules, the posterior probabilities are:

$$P(Y = j | \vec{X} = \mathbf{x}) = \frac{f(\mathbf{x} | Y = j) \cdot P(Y = j)}{f(\mathbf{x})} \propto f_j(\mathbf{x}) \pi_j$$

Therefore, they are model-based Bayes Classifiers with classification rule:

$$\hat{j} = \operatorname{argmax}_j \underbrace{f_j(\mathbf{x})}_{\text{likelihood}} \cdot \underbrace{\pi_j}_{\text{prior prob}}$$



Part A

Bayes Classifier - Linear Discriminant Analysis (LDA) (Same covariance matrix)

- Assumes normally distributed data
 - The components' distributions are all multivariate Gaussian

$$f_j(\mathbf{x}) = \frac{1}{(2\pi)^{d/2} |\Sigma_j|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu_j)^T \Sigma_j^{-1} (\mathbf{x}-\mu_j)}, \quad \forall j = 1, \dots, c$$

- Assumes a class-specific mean value vector
$$\mu_1, \mu_2, \mu_3$$
- The Gaussians for each class (1,2 and 3) share the same covariance matrix
$$\Sigma_1 = \Sigma_2 = \Sigma_3 = \Sigma$$

- The different distributions of the components are basically shifted versions of each other
 - Same covariance matrix with different centers

When these assumptions hold, then LDA approximates the Bayes classifier very closely!

- LDA is also known to achieve good performances when these assumptions do not hold and a common covariance matrix among groups and normality are often violated.



Part A

Bayes Classifier - Linear Discriminant Analysis (LDA) (Same covariance matrix)

Some mathematical analysis about the LDA Classifier:

- Given training data, it uses maximum Likelihood method for parameter estimation

$$\hat{\mu}_j = \frac{1}{n_j} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i$$

$$\hat{\Sigma} = \frac{1}{n - c} \sum_{j=1}^c \sum_{\mathbf{x}_i \in C_j} (\mathbf{x}_i - \hat{\mu}_j)(\mathbf{x}_i - \hat{\mu}_j)^T$$

- Thus, the classification rule is:

$$\hat{j} = \operatorname{argmax}_j \mathbf{x}^T \hat{\Sigma}^{-1} \hat{\mu}_j - \frac{1}{2} \hat{\mu}_j^T \hat{\Sigma}^{-1} \hat{\mu}_j + \log \hat{\pi}_j, \text{ where: } \hat{\pi}_j = \frac{n_j}{n}$$

- The decision boundary formula:

$$\mathbf{x}^T \Sigma^{-1} (\mu_j - \mu_\ell) = \frac{1}{2} (\mu_j^T \Sigma^{-1} \mu_j - \mu_\ell^T \Sigma^{-1} \mu_\ell) + \log \frac{\pi_\ell}{\pi_j}$$

- This is a hyperplane with normal vector $\Sigma^{-1}(\mu_j - \mu_\ell)$, showing that we expect to have linear decision boundaries in our classification later (between each pair of training classes)



Part A

Bayes Classifier - Quadratic Discriminant Analysis (QDA) (Different covariance matrix)

- Assumes normally distributed data **(Same as LDA)**
 - The components' distributions are all multivariate Gaussian

$$f_j(\mathbf{x}) = \frac{1}{(2\pi)^{d/2} |\Sigma_j|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu_j)^T \Sigma_j^{-1} (\mathbf{x}-\mu_j)}, \quad \forall j = 1, \dots, c$$

- Assumes a class-specific mean value vector **(Same as LDA)**
 μ_1, μ_2, μ_3
- The Gaussians for each class (1, 2 και 3) has its own covariance matrix
 $\Sigma_1, \Sigma_2, \Sigma_3$
 - **(Different from LDA)**

When these assumptions hold, then QDA approximates the Bayes classifier very closely!



Part A

Bayes Classifier - Quadratic Discriminant Analysis (QDA) (Different covariance matrix)

Some mathematical analysis about the QDA Classifier:

- Given training data, it uses maximum Likelihood method for parameter estimation

$$\hat{\boldsymbol{\mu}}_j = \frac{1}{n_j} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i$$

$$\hat{\boldsymbol{\Sigma}}_j = \frac{1}{n_j - 1} \sum_{\mathbf{x}_i \in C_j} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_j)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_j)^T$$

- Thus, the classification rule is:

$$\hat{j} = \operatorname{argmax}_j \log \hat{\pi}_j - \frac{1}{2} \log |\hat{\boldsymbol{\Sigma}}_j| - \frac{1}{2} (\mathbf{x} - \hat{\boldsymbol{\mu}}_j)^T \hat{\boldsymbol{\Sigma}}_j^{-1} (\mathbf{x} - \hat{\boldsymbol{\mu}}_j), \text{ where: } \hat{\pi}_j = \frac{n_j}{n}$$

- The decision boundary formula:

$$\log \pi_\ell - \frac{1}{2} \log |\boldsymbol{\Sigma}_\ell| - \frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_\ell)^T \boldsymbol{\Sigma}_\ell^{-1} (\mathbf{x} - \boldsymbol{\mu}_\ell)$$

- This shows that we expect to have quadratic boundaries (between each pair of training classes) in our classification later

Part A

Bayes Classifier

LDA vs QDA (Mean Classification Error)

Code for LDA and QDA (accordingly) classification and result:

```
1 model_LDA = LinearDiscriminantAnalysis()  
2 model_LDA.fit(X_train, y_train)  
3  
4 y_pred_LDA = model_LDA.predict(X_test).astype(int)  
5  
6 meanError_LDA = zero_one_loss(y_test, y_pred_LDA)  
7 print("Bayes - Same covariance matrix - Mean Classification Error =", meanError_LDA)
```

Bayes - Same covariance matrix - Mean Classification Error = 0.22857142857142854

```
1 model_QDA = QuadraticDiscriminantAnalysis()  
2 model_QDA.fit(X_train, y_train)  
3  
4 y_pred_QDA = model_QDA.predict(X_test).astype(int)  
5  
6 meanError_QDA= zero_one_loss(y_test, y_pred_QDA)  
7 print("Bayes - Different covariance matrix - Mean Classification Error =", meanError_QDA)
```

Bayes - Different covariance matrix - Mean Classification Error = 0.18571428571428572

Model Initialization

Model Training on train dataset

Model predictions on test dataset

Mean classification error metric calculation by comparing real and predicted label of train dataset

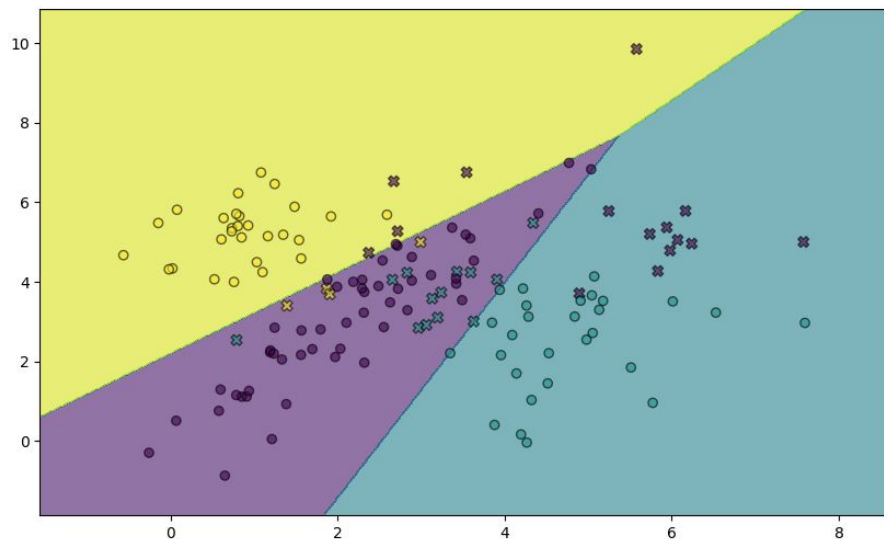
We used the same test set!

Part A

Bayes Classifier

LDA vs QDA (Decision Boundaries)

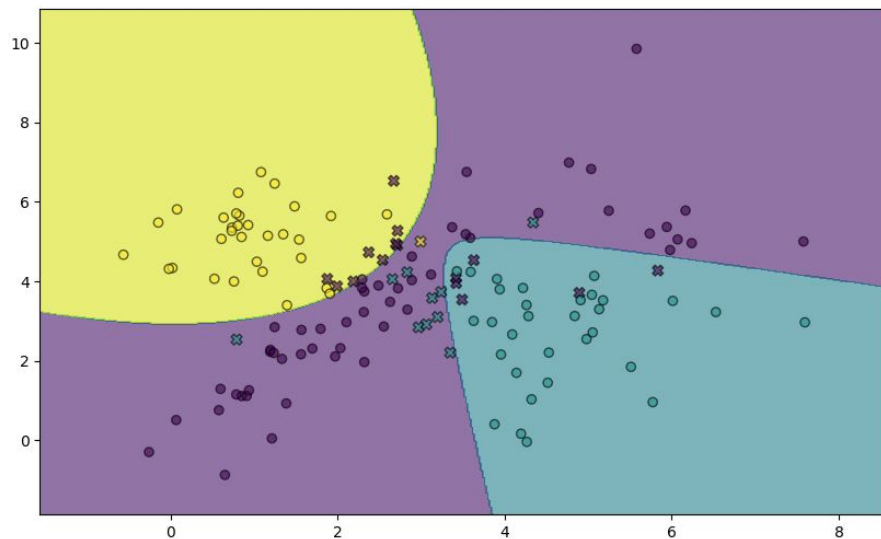
Bayes Classifier
Same covariance matrix for all classes
Linear Discriminant Analysis (LDA)
(Plotting test data and the decision boundaries for each class)



Dataset markers
○ Test (Correctly Classified)
✕ Test (Wrongly Classified)

True Label Color
● 1 ● 2 ● 3

Bayes Classifier
Different covariance matrix for each class
Quadratic Discriminant Analysis (QDA)
(Plotting test data and the decision boundaries for each class)



Dataset markers
○ Test (Correctly Classified)
✕ Test (Wrongly Classified)

True Label Color
● 1 ● 2 ● 3



Part A

Bayes Classifier

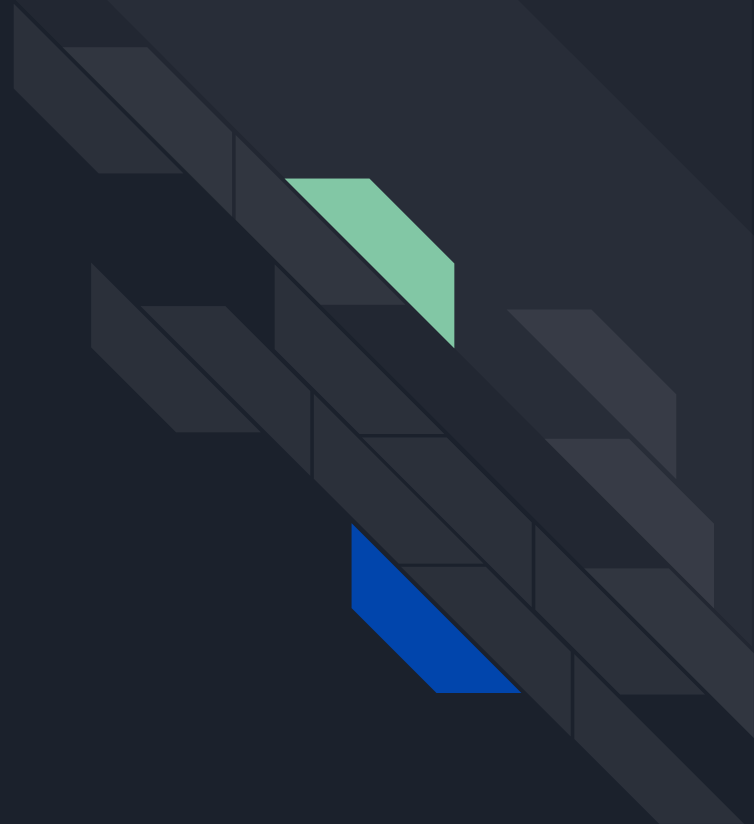
LDA vs QDA (Conclusions)

- QDA gives a better result than the LDA
 - $e_{LDA} = 0.2285714 > 0.18571428 = e_{QDA}$
- From the decision boundaries, we can see that:
 - LDA can be trained only with linear decision boundaries (between each class)
 - QDA can be trained with quadratic boundaries (between each class)
 - Therefore, QDA is more flexible
- The result is expected since:
 - The higher the dimension of the dataset (number of predictors), QDA has to calculate more parameters
 - This can lead to problems such as bigger variance or/and overfitting
 - Here we have an only 2-D dataset
 - Data with more predictors can more easily be linearly separated
- When the dataset's classes (here they are 3) have very different covariance matrices
 - LDA suffers from high biases
 - Here, the two classifiers (LDA and QDA) have similar error results
 - Therefore, the dataset's classes don't have very different covariance matrices
- In general, it comes down to a bias-variance trade-off

Part B

k - NN Classifier

- Experiment for different number of neighbors (k)
 - $k = 1, 2, 3, \dots, 10$





Part B

k - NN Classifier

General Information

- k - NN is also a Bayes classifier
 - BUT it is nonparametric (**Non-parametric does not mean no parameters!**)
 - No fixed number of parameters
 - Parameters grow with the number of training data points
- **GENERAL IDEA:**
 - The value of the target function for a new query is estimated from the known value(s) of the nearest training example(s)

1. Find k examples $\{\mathbf{x}^{(i)}, t^{(i)}\}$ closest to the test instance \mathbf{x}
2. Classification output is majority class

$$y = \arg \max_{t^{(z)}} \sum_{r=1}^k \delta(t^{(z)}, t^{(r)})$$

- Distance typically defined to be Euclidean: $\|\mathbf{x}^{(a)} - \mathbf{x}^{(b)}\|_2 = \sqrt{\sum_{j=1}^d (x_j^{(a)} - x_j^{(b)})^2}$
- Does NOT explicitly compute decision boundaries
 - But can be inferred

Part B

k - NN Classifier

k = 1, 2, 3, ..., 10 (Mean Classification Error)

```
1 # Run training process for every k = 1, 2, ..., 10
2 k_min = 1
3 k_max = 10
4
5 meanError_knn = np.zeros((k_max - k_min + 1,), dtype=np.float64)
6 fig, ax = plt.subplots(nrows=5, ncols=2, figsize=(9, 18))
7 ax = ax.T.flatten()
8 for k in range(k_min, k_max + 1):
9     # Train model
10    model_knn = KNeighborsClassifier(n_neighbors=k)
11    model_knn.fit(X_train, y_train)
12
13    # Get predicted labels for test set
14    y_pred_knn = model_knn.predict(X_test).astype(int)
15
16    # Calculate error
17    meanError_knn[k - 1] = zero_one_loss(y_test, y_pred_knn)
18
```

We used the same test set as before!

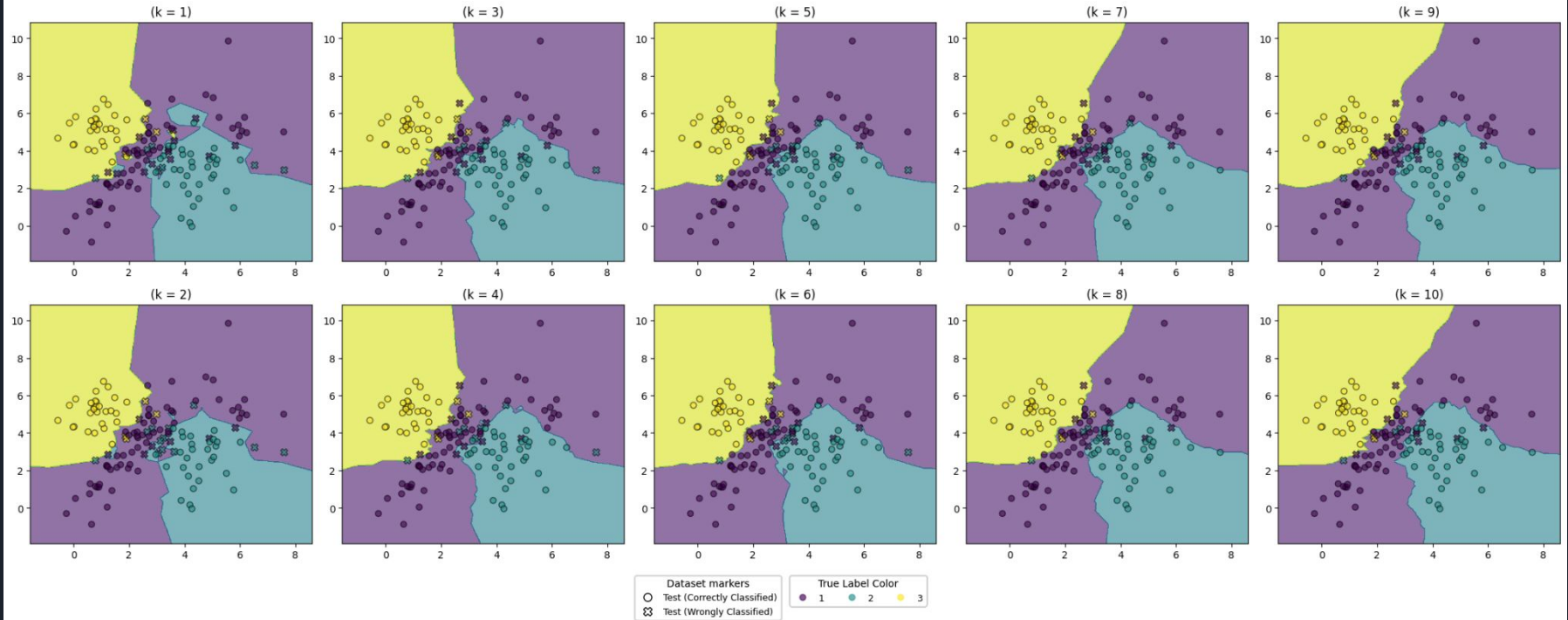
Number of Neighbors (k)	Mean Classification Error
1	0.164286
2	0.164286
3	0.135714
4	0.135714
5	0.142857
6	0.142857
7	0.121429
8	0.114286
9	0.107143
10	0.107143

Part B

k - NN Classifier

k = 1, 2, 3, ..., 10 (Decision Boundaries)

k - NN Classifier
For every k = 1, 2, ..., 10
(Plotting test data and the decision boundaries for each class)



Part B

k - NN Classifier

k = 1, 2, 3, ..., 10 (Conclusions)

- The bigger the value of k, the smaller the value of the mean classification error tends to get
 - However, we can see in some instances that the mean classification error rises
- | | |
|---|----------|
| 4 | 0.135714 |
| 5 | 0.142857 |
| 6 | 0.142857 |
| 7 | 0.121429 |
- This occurs when the value of k is small, the model “selects” only the nearest values near the test sample
 - Thus creating a very complex decision boundary
 - This leads to problems such as overfitting
 - The model can’t generalize good enough to “produce” many accurate predictions
 - The bigger the value of k, the smoother the decisions boundaries get (compare k=1 and k = 10 plots)
 - Better generalization
 - However, if k gets very big, we might end up “looking” at training samples that aren’t neighbors
 - For k = 10, we get the smaller mean classification error
 - It’s incorrect to say that this is the best model for the dataset
 - We need a validation method for tuning (such as Cross Validation)
 - Rule of thumb: $k < \sqrt{n}$, where n is the number of training samples



Part B

k - NN Classifier

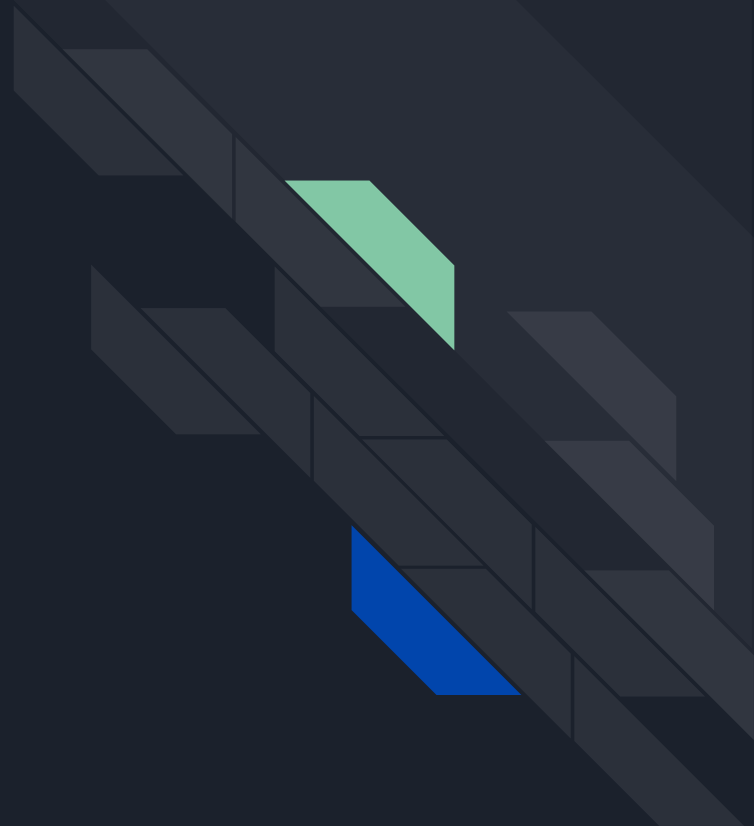
k = 1, 2, 3, ..., 10 (Comparison with the Bayes Classifiers of Part A)

- k - NN, in general, performs better than LDA and QDA
 - Even the “worst” k-NN model performs better than LDA and QDA
$$e_{\text{LDA}} = 0.2285714 > 0.18571428 = e_{\text{QDA}} > 0.164286 = e_{1\text{-NN}}$$
- This might happen because of the ability of the k-NN to create decision boundaries based on the training data themselves and not on some linear/quadratic equation
- We can see that the decision regions are not separated by a linear / quadratic line
 - A lot of our samples are near each other and they have very different labels
 - This produces a complex geometry in the decision boundaries plots
- If we did a search for k values greater than 10, we might have gotten a smaller error value
- However, our accuracy is almost excellent (almost 90% for k = 10)

Part C

SVM Classifier

- Experiment for different kernel functions
 - Linear
 - Radial Basis Function (RBF)
 - Experiment with RBF's hyperparameters





Part C

SVM Classifier

General Information

- Support Vector Machines (SVMs) aim to find a hyperplane that effectively separates the classes in their training data by maximizing the margin between the outermost data points of each class.
 - Achieved by finding the best weight vector that defines the decision boundary hyperplane and minimizes the losses for misclassified samples
 - The training datapoints which are at the minimum distance to the hyperplane are called **Support Vectors**
- In its most simple type, SVM doesn't support multiclass classification natively
 - It supports binary classification and separating data points into two classes
 - For multiclass classification, the same principle is utilized after breaking down the multiclassification problem into multiple binary classification problems
 - We used the **One-vs-Rest approach**
 - If we have an N (Here $N = 3$) class problem, we (internally) create N SVMs
 - SVM #1 learns “class_output = 1” vs “class_output != 1”
 - SVM #2 learns “class_output = 2” vs “class_output != 2”
 - ...

Part C

SVM Classifier - Linear Kernel Function (Decision Boundaries and Mean Classification Error)

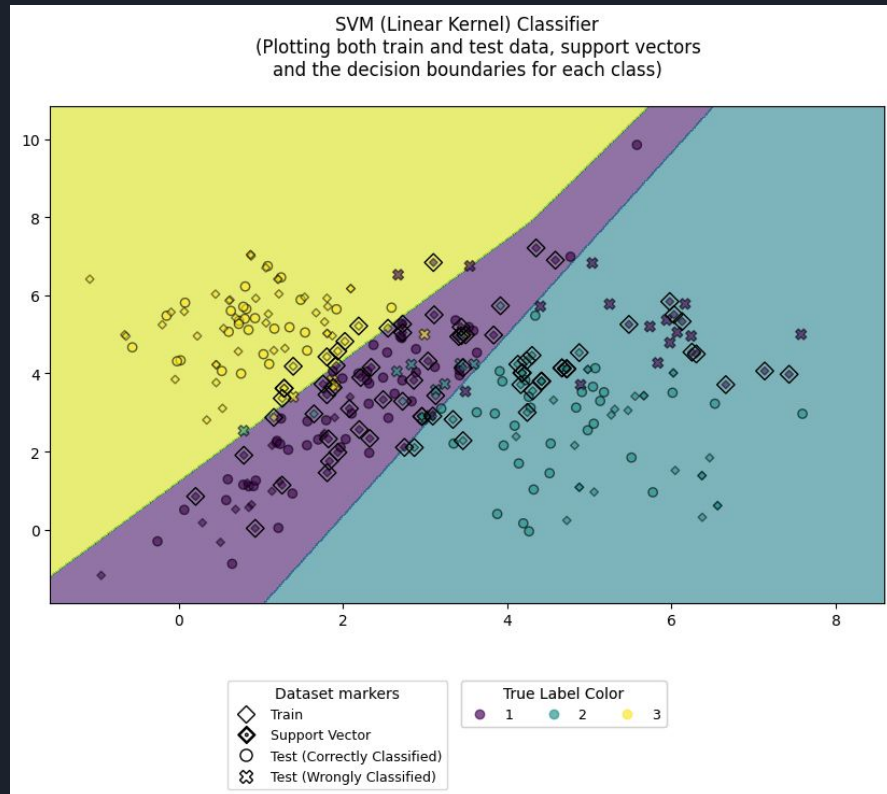
- Linear kernel is the dot product of the input samples
 - Defined as: $K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1^\top \mathbf{x}_2$
- It is then applied to any combination of two data points (samples) in the dataset
- The dot product of the two points determines the cosine similarity between both points
 - The higher the value, the more similar the points are

```
1 # Train model
2 model_linearSVM = svm.SVC(kernel='linear', random_state=42)
3 model_linearSVM.fit(X_train, y_train)
4
5 # Get predicted labels for test set
6 y_pred_linSVM = model_linearSVM.predict(X_test).astype(int)
7
8 # Calculate error
9 meanError_linSVM = zero_one_loss(y_test, y_pred_linSVM)
10 print("Linear SVM - Mean Classification Error =", meanError_linSVM)
```

Linear SVM - Mean Classification Error = 0.19999999999999996

We see that:

- The hyperplanes are straight lines
- Due to the lack of expressivity of the linear kernel, the trained classes do not perfectly capture the training data





Part C

SVM Classifier - RBF Kernel Function Hyperparameters Experiment

- The Radial Basis Function (RBF) kernel (a.k.a Gaussian kernel) measures similarity between two data points in infinite dimensions and then approaches classification by majority vote
 - Defined as:

$$K(\mathbf{x}_1, \mathbf{x}_2) = \exp(-\gamma \cdot \|\mathbf{x}_1 - \mathbf{x}_2\|^2)$$
- There are two hyperparameters
 - The normalization parameter C (*Not visible in the function above*)
 - Trades off correct classification of training examples against maximization of the decision function's margin
 - For larger values of C , a smaller margin will be accepted if the decision function is better at classifying all training points correctly
 - A lower C will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy
 - The kernel's coefficient γ
 - Defines how far the influence of a single training example reaches
 - Low values = far
 - High values = close
 - Inverse of the radius of influence of samples selected by the model as support vectors

Part C

SVM Classifier - RBF Kernel Function

Hyperparameters Experiment (Mean Classification Error Heatmap)

We create a heatmap for the mean classification error to see if there are any patterns on the error calculation

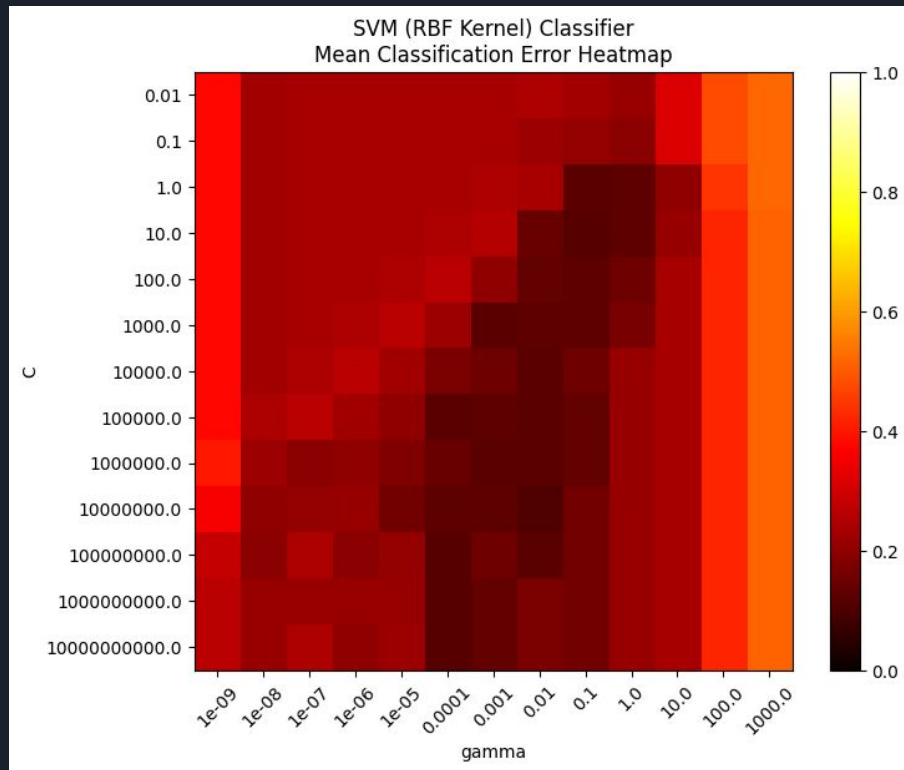
- On the **same test set** as before!!

We make an extended grid search for values spaced evenly on a log scale:

- C in $[10^{-9}, 10^3]$ with 13 points
- γ in $[10^{-2}, 10^{10}]$ with 13 points

```
1 C_range = np.logspace(-2, 10, 13)  
2 gamma_range = np.logspace(-9, 3, 13)
```

- $13 \times 13 = 169$ total error calculations



Part C

SVM Classifier - RBF Kernel Function

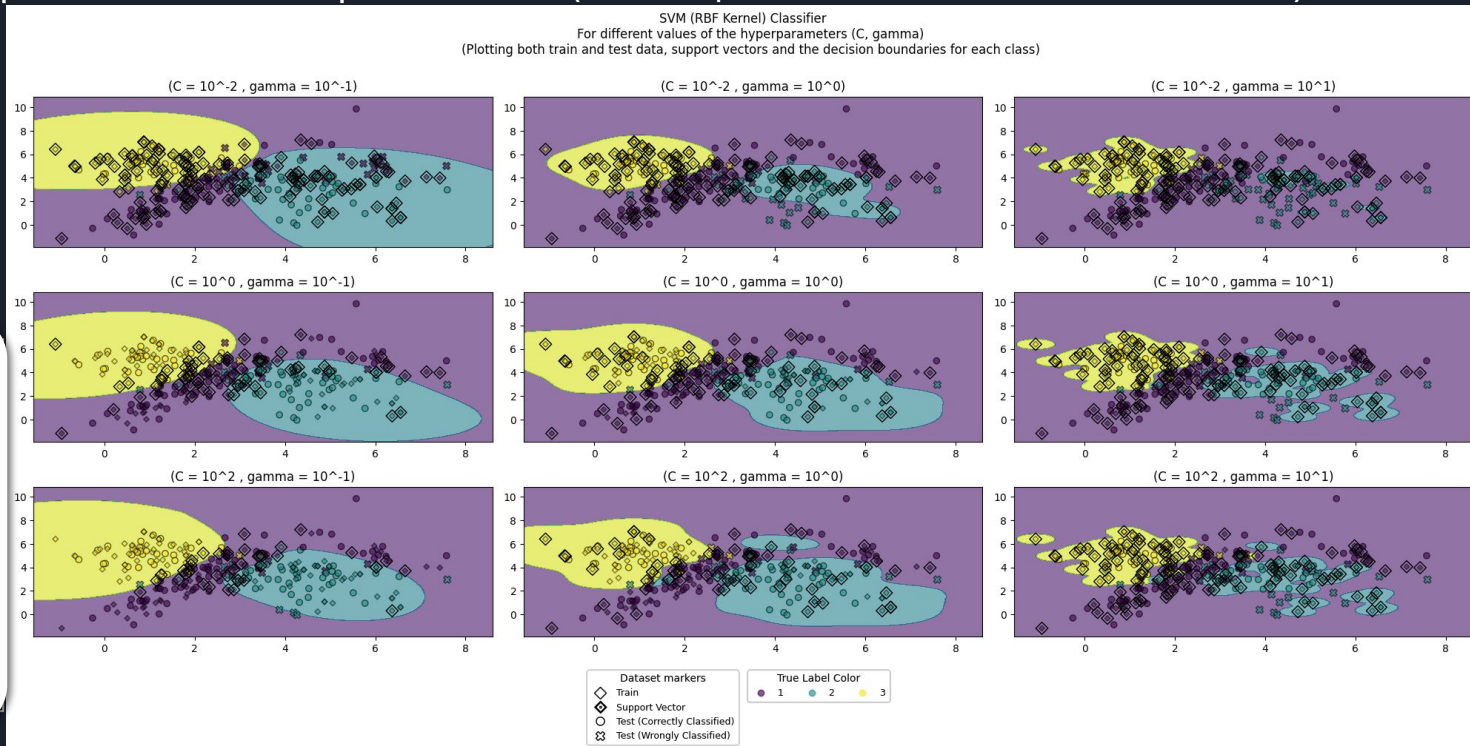
Hyperparameters Experiment (Some plots and mean error values)

We did some plots for a smaller range of hyperparameter values

- $C \in \{10^{-2}, 1, 10^2\}$
- $\gamma \in \{10^{-1}, 1, 10^1\}$

```
2 C_range = [1e-2, 1, 1e2]
3 gamma_range = [1e-1, 1, 1e1]
```

C	gamma	Mean Classification Error
0.010000	0.100000	0.228571
0.010000	1.000000	0.214286
0.010000	10.000000	0.314286
1.000000	0.100000	0.121429
1.000000	1.000000	0.128571
1.000000	10.000000	0.200000
100.000000	0.100000	0.128571
100.000000	1.000000	0.150000
100.000000	10.000000	0.235714



We used the same test set as before!



Part C

SVM Classifier - RBF Kernel Function

Hyperparameters Experiment (Conclusions)

From the plots we saw that:

- Model is very sensitive to the γ value
 - If γ is too large, the radius of the area of influence of the support vectors only includes the support vector itself
 - No amount of regularization with C seems to be able to prevent overfitting
 - When gamma is very small, the model is too constrained
 - Cannot capture the complexity / “shape” of the data
 - For $\gamma = 0.1$ we generally get the best results (minimum mean classification error)
 - However, the decision boundaries are very close to the testing data
 - Overfitting would occur if the testing and training sets weren’t closely correlated

From the heatmap we saw that:

- We get good models on a diagonal of C and γ , because:
 - Lower γ values \rightarrow Smooth models
 - We want more complexity \rightarrow Increase the importance of classifying each point correctly
 - Increase the importance of classifying each point correctly \rightarrow larger C values
 - See that $e_{\gamma=1, C=1} = e_{\gamma=0.1, C=100}$
- Since the dataset has very “crowded” samples, we can easily find a good pair of values
- For some intermediate values of γ we get equally performing models when C becomes very large
 - The set of support vectors does not change anymore
- Lower C values generally lead to more support vectors



Part C

SVM Classifier

(Comparison between kernels and between the classifiers before)

Comparing the two kernels between them see we that:

- The RBF kernel performs better (for correctly-tuned values of hyperparameters) than the linear kernel
 - $e_{\text{linear SVM}} = 0.1999 > 0.12857 = e_{\text{RBF SVM}\{C=1, \gamma=1\}}$

Comparing the SVM with the k - NN and Bayes Classifiers:

- By comparing the mean classification errors linear SVM gives better results than k - NN and Bayes
- As expected, RBF SVM gives the best results out of all the classifiers
 - Because of its complexity
 - However, SVM does the assumption that there is a hyper-plane that can separate the data
 - A very limiting assumption but it works in our dataset
- With correct parameter choice on the SVM → Almost excellent classifier (~90% accuracy)
- With wrong parameter choice on the SVM → SVM classifier becomes worse than the LDA
- The RBF SVMs with small values of mean classification error:
 - The decision regions of the classes 1 and 3 seem to be very close to the data
 - If our dataset was more generalized, we could have problems such as overfitting
 - k - NN might have been the better choice

Part D





Introduction

Part D

- Explore an unknown high dimension dataset
- Find the best possible ML model for this dataset using the same training subset and train it
- Export the labels of another dataset (the labels aren't provided)

Models are mostly created and trained using the  scikit-learn python library

Furthermore, we also trained some Neural Networks using the  PyTorch python library

For comparing the models and deciding which one to keep to train and export the labels, we use the mean classification error metric (Zero-One Classification Loss) and keep the one with the greater value

$$e = \frac{\sum \text{Wrongly classified test samples}}{\sum \text{Test samples}} = (1 - \text{accuracy})$$

We will work on the datasetC.csv dataset to find the optimal model (provided by the assignment)

Then, using the best trained model, we will export the labels of the datasetCTest.csv dataset

- We used GitHub and Git to import the dataset into our  Google Colab notebook

Import, Exploration/Visualization and Preprocessing

(1) `datasetC.csv`

(7)

datasetC.csv

	X																			Y	
	0	1	2	3	4	5	6	7	8	9	...	391	392	393	394	395	396	397	398	399	400
0	0.131550	0.840360	-1.12770	-0.51572	1.738600	1.84650	0.707460	-1.02400	-0.34507	0.847810	...	0.67800	1.58660	-0.037355	0.093535	0.783140	0.014524	0.25725	-0.76285	0.33389	5
1	-0.242500	0.511610	-1.04290	-0.76597	1.784600	1.13210	2.051500	1.49950	0.01855	1.099800	...	-0.53046	0.51418	0.743820	-0.795280	-1.714600	-0.248340	0.36943	0.42895	-0.83821	4
2	0.225580	0.257230	1.51950	1.00860	-0.11470	0.66979	0.405690	0.21703	0.62943	0.016552	...	-0.22967	0.63654	-0.063802	-0.478290	0.901530	0.748410	-0.10565	-0.88156	0.75886	1
3	0.582160	0.545290	-0.37667	-0.42480	0.940330	1.32610	0.888940	0.34529	-0.81031	0.844940	...	-0.89579	-0.53431	0.902910	2.804000	-0.704650	0.806900	-0.75800	1.04360	1.60830	2
4	0.006266	0.562850	2.20540	0.98011	0.016941	1.16290	1.547400	-0.25004	-0.29866	-1.402500	...	-0.26117	1.74760	0.690260	0.807960	-1.705700	-0.291630	0.15721	1.52700	-0.44787	1
...
4995	-1.500800	-0.424520	1.09190	1.20690	0.090575	1.75910	-0.050586	0.54109	-0.19073	-0.785740	...	-0.67415	0.48537	0.311380	-0.416360	-0.134820	-0.054313	-0.45146	-0.23687	-0.22282	1
4996	1.569400	-0.147500	1.55880	-0.13371	-2.410400	0.34831	-0.935490	1.14710	0.56856	0.117950	...	0.84073	-0.54290	-0.634520	-0.360470	-0.119520	-0.106600	0.37003	-0.52111	0.91101	5
4997	2.031300	0.138280	1.47530	-0.40413	-0.449720	0.78086	-0.765770	-0.87269	0.45053	0.877180	...	-0.48120	1.08700	-0.814570	-1.150100	-0.019056	0.985370	0.59845	-0.21938	0.40070	2
4998	-0.471720	-0.457780	0.86395	0.31563	0.817330	0.98600	0.756050	-0.50074	1.500770	-1.883000	...	1.47090	1.49430	0.602310	-0.119280	-0.744420	0.559260	1.84720	0.11341	1.76620	3
4999	0.746870	0.025161	1.35570	-1.10050	-0.184320	1.02920	0.962060	0.95188	-0.22038	-0.112230	...	0.65798	0.31885	0.048799	0.601560	-0.741670	-0.494040	0.56348	0.21997	0.42807	4

5000 rows \times 401 columns

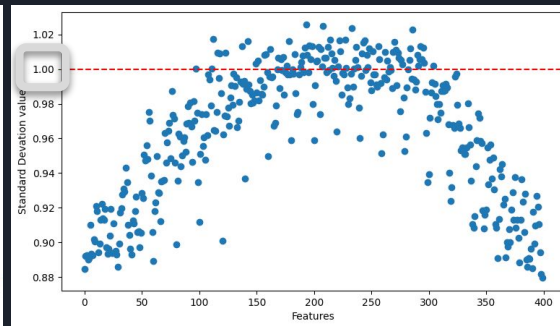
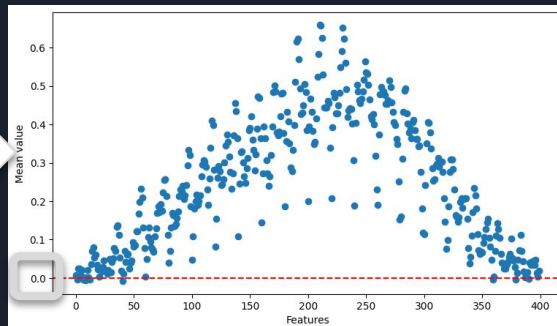
Features' Mean Values

Features' Standard Deviations

- A mean value near 0
- A standard deviation near 1

So, we seem to have a **normalized dataset**

Quite important as normalization gives equal weights / importance to each feature so that none of them "tilts" our models performance in one direction just simply because of large numbers.

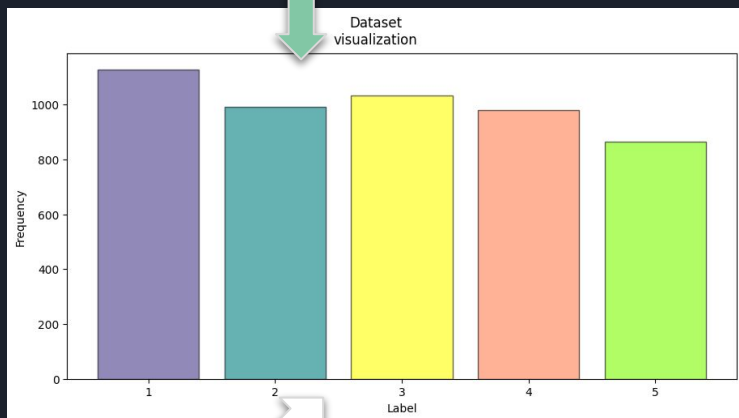


Dataset

Import, Exploration/Visualization and Preprocessing (2) (Split)

datasetC.csv

Dataset
visualization

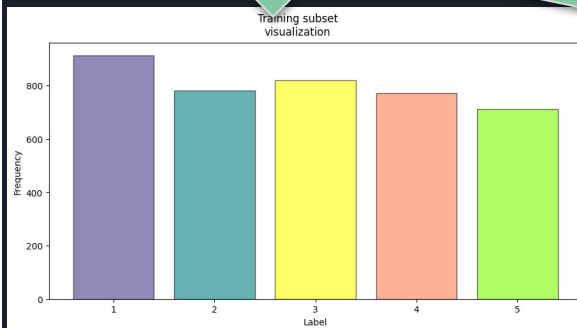


Split Dataset

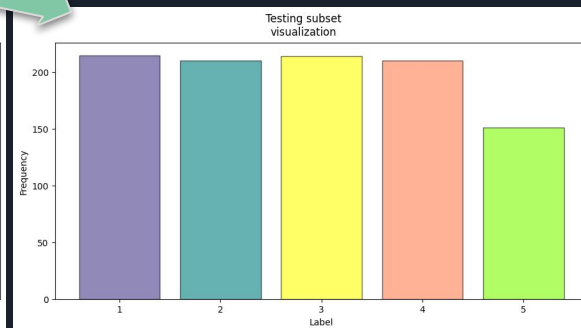
- 80% Training
- 20% Testing

```
1 split_scale = 0.2 # Split into training and test sets with a 80% - 20% ratio
2
3 X_train, X_test, y_train, y_test = train_test_split(X, y,
4                                                    test_size=split_scale,
5                                                    shuffle=True,
6                                                    random_state=42)
```

Training subset
visualization



Testing subset
visualization



We see that we have:

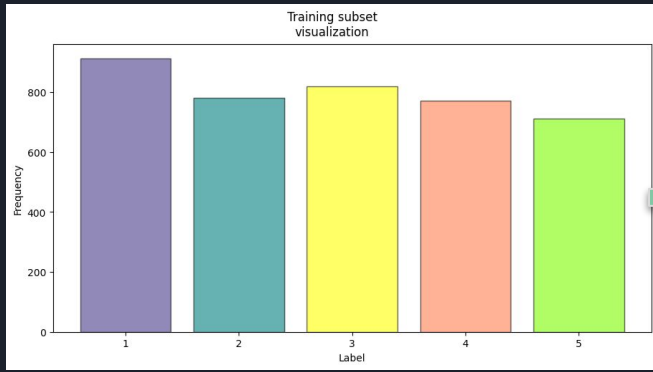
- 5 Classes
- Imbalanced (labels) dataset

We splitted this way so we can have both:

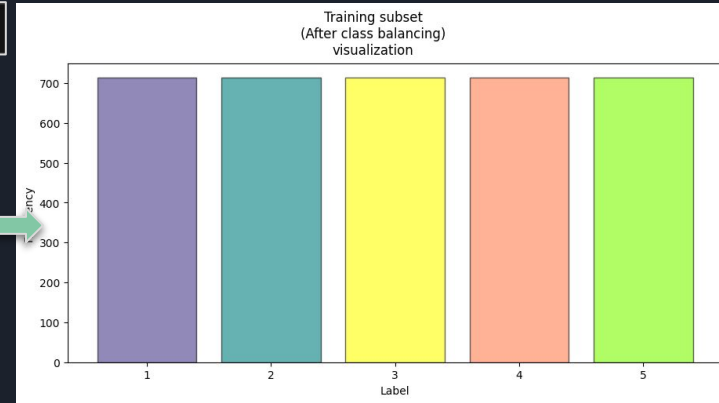
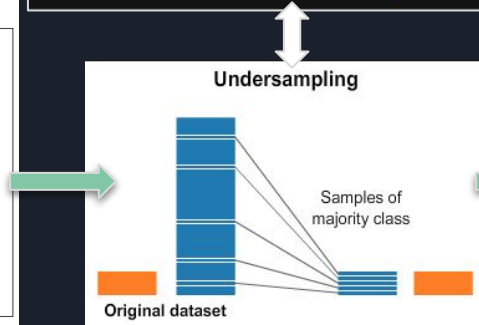
- Good amount of data to train our classifiers
- Good amount of data to test our classifiers and get a representing result

Dataset

Import, Exploration/Visualization and Preprocessing (3) (Class Balancing)



```
1 rus = RandomUnderSampler(random_state=42)
2 X_train_balanced, y_train_balanced = rus.fit_resample(X_train, y_train)
```



- Training dataset is **NOT** balanced
 - All labels are not on the same (or almost) level / height
- This can create problems during the training and testing process
 - Biases (Model might end up performing good only for some classes)

We fix this issue using Oversampling / Undersampling techniques
We choose to **Undersample** the dataset because:

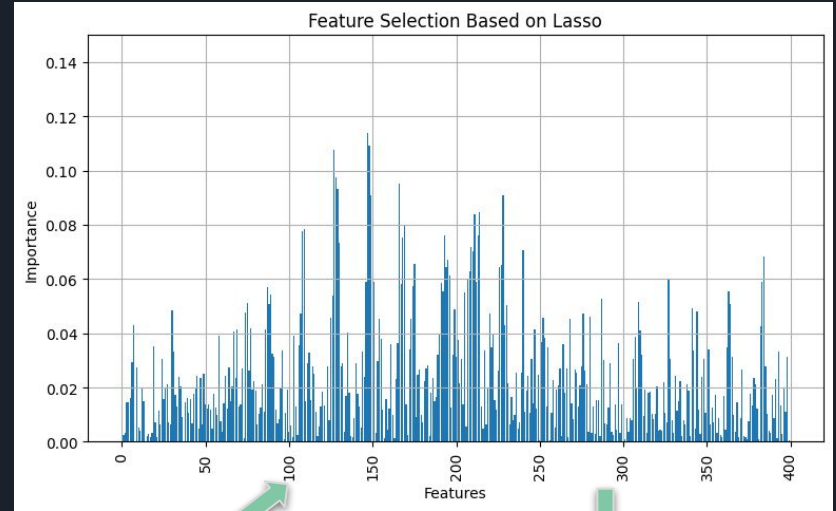
- Large Dataset
 - 4000 (0.8 x 5000) samples in the training dataset
 - We can afford to lose some samples
 - Oversampling might lead to overfitting
 - We want to avoid because we want to export labels of another dataset

We keep the testing dataset as is so we can have a realistic indication of the performance of our models!

Dataset

Import, Exploration/Visualization and Preprocessing (3) (Dimensionality Reduction)

- Dimension reduction can improve classifiers' accuracies
 - Remove redundant features
 - They can even lead to overfitting and lack of generalization
- By selecting the most important features, the model can "focus" on the most important information
- We used the **LASSO** method
 - Last Absolute Shrinkage and Selection Operator
 - Frequently used in regression
 - It adds a penalty in the loss function
 - Based on the coefficients' absolute values
 - Shrink the least important features' coefficients to 0



Firstly, we used CV to find the best parameter α

- Constant that multiplies with penalty L_1
- Controlling regularization

```
1 # Parameters to be tested on GridSearchCV
2 params = {"alpha":np.arange(0.00001, 10, 500)}
3
4 # Number of Folds and adding the random state for replication
5 kf = KFold(n_splits=5, shuffle=True, random_state=42)
6
7 # Initializing the Model
8 lasso = Lasso()
9
10 # GridSearchCV with model, params and folds.
11 lasso_cv = GridSearchCV(lasso, param_grid=params, cv=kf)
12 lasso_cv.fit(X, y)
13
14 print("Best Params {}".format(lasso_cv.best_params_))
Best Params {'alpha': 1e-05}
```

Re-train
a LASSO model

```
1 feature_subset = np.array(range(numOfFeatures, [bestLasso_coef>0.001]))
2
3 numOfImportantFeatures = np.size(feature_subset)
4
5 print("Number of important features:", numOfImportantFeatures)
6 print("Number of features dropped:", numOfFeatures-numOfImportantFeatures)
```

Number of important features: 392
Number of features dropped: 8

Maybe we can't apply feature selection?
We will test if it increase the accuracy later

Part D

Testing different models

(1)

- Since we have a high-dimension dataset, we think our data might be easily linearly separable
 - Therefore, the simplicity of some classifiers might lead to better generalization than others
 - Naive Bayes
 - Linear SVM
 - LDA (Linear Discriminant Analysis)

- Tryouts:

- Naive Bayes

- Using only the splitted dataset →

```
1 nb = GaussianNB()  
2  
3 nb.fit(X_train, y_train)  
4  
5 nbScore = nb.score(X_test, y_test)  
6 print(nbScore)
```

0.804

- Using the balanced dataset →

```
1 nb = GaussianNB()  
2  
3 nb.fit(X_train_balanced, y_train_balanced)  
4  
5 nbScore = nb.score(X_test, y_test)  
6 print(nbScore)
```

0.807

BEST MODEL SO FAR

- Using the (feature) reduced dataset →

```
1 nb = GaussianNB()  
2  
3 nb.fit(X_train[:, feature_subset], y_train)  
4  
5 nbScore = nb.score(X_test[:, feature_subset], y_test)  
6 print(nbScore)
```

0.805

Part D

Testing different models (2)

- Indeed, we got some pretty good results with the Naive Bayes Classifier
 - ~ 80% accuracy
 - Despite the simplicity of the classifier
- The Naive Bayes classifier assumes that the predictors **are conditionally independent**, or unrelated to any of the other feature in the model
 - We get a hint that we might have conditionally independent features in our dataset
- We see that the (dimension) redacted dataset doesn't improve the accuracy by a lot!

- Tryouts:
 - LDA (Linear Discriminant Analysis)

- Using the balanced dataset →

```
1 # Initialize and train model
2 model_LDA = LinearDiscriminantAnalysis()
3 model_LDA.fit(X_train_balanced, y_train_balanced)
4
5 model_LDA.score(X_test, y_test)
```

0.807

- Using the (feature) redacted dataset →

```
1 # Initialize and train model
2 model_LDA = LinearDiscriminantAnalysis()
3 model_LDA.fit(X_train_balanced[:, feature_subset], y_train_balanced)
4
5 model_LDA.score(X_test[:, feature_subset], y_test)
```

0.802

Part D

Testing different models (3)

- We get similar results
 - We get a hint that our features might be linearly (or almost linearly) correlated in some way
- We see that the dataset with the reduced number of features **LOWERS** the accuracy
 - Not worth the risk of using this dataset
 - We continue using all the features of the dataset (but the balanced one)

- Tryouts:

- **Linear SVM** →

```
1 # Initialize and train model
2 model_linearSVM = svm.SVC(kernel='linear', random_state=42)
3 model_linearSVM.fit(X_train_balanced, y_train_balanced)
4
5 model_linearSVM.score(X_test, y_test)
```

0.769

- We didn't "beat" our previous accuracy record
 - **RBF Kernel SVM**
 - We run a small experiment with the RBF kernel's hyperparameters
 - **Similar to Part C**
 - We run a grid search and plot the accuracies in a heatmap to search for any patterns

```
1 C_range = np.logspace(-2, 10, 13)
2 gamma_range = np.logspace(-9, 3, 13)
```

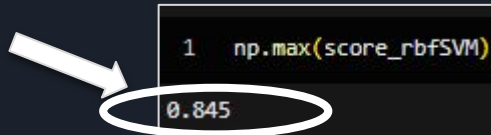
Part D

Testing different models (4)

- For $\gamma \geq 10$ we get a sudden drop in accuracy
- For $\gamma < 10$, we get similar results
- C doesn't seem to affect the accuracy much
 - Except when $\gamma = 0.1$

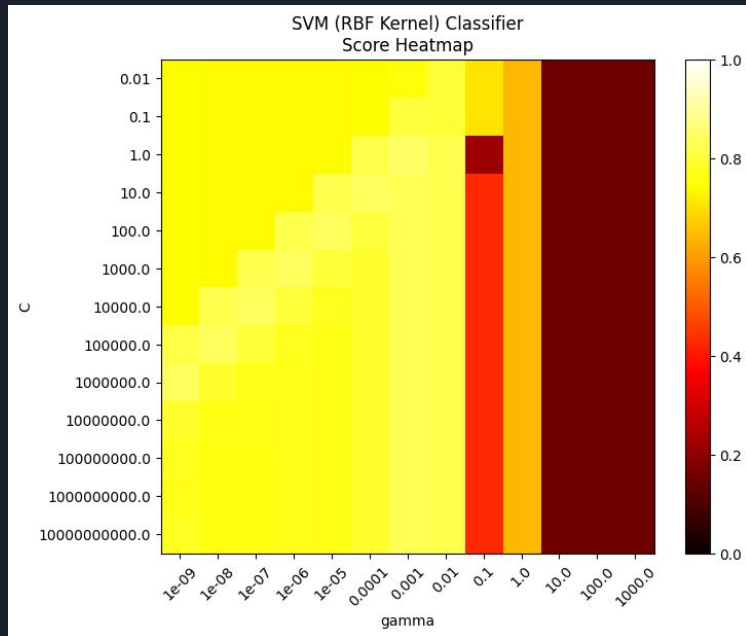
In general, we seem to get pretty good results

BEST MODEL SO FAR



We even beat our previous best accuracy!!

- However, we think we can find an even better classifier that gives a better accuracy on the test set
- If we don't, we will come back to the RBF kernel SVM for further research



Part D

Testing different models (5)

- Tryouts:

- k - NN Classifier

- We run a small experiment with the number of neighbors (k)
 - **Similar to Part B**
 - We got the accuracies for $k = 1, 2, 3, \dots, 30$

```
1 np.max(score_knn)
```

0.774

- We seem to be really far of our best accuracy so far so we don't do any more research on the k - NN

- Gaussian Process →

```
1 gp = GaussianProcessClassifier(random_state=42)
2
3 gp.fit(X_train_balanced, y_train_balanced)
4
5 gp.score(X_test, y_test)
```

0.151

- QDA (Quadratic Discriminant Analysis) →

```
1 model_QDA = QuadraticDiscriminantAnalysis()
2
3 model_QDA.fit(X_train_balanced, y_train_balanced)
4
5 model_QDA.score(X_test, y_test)
```

0.424

Part D

Testing different models (6)

- Random Forest →

```
1 rf = RandomForestClassifier(random_state=42)
2
3 rf.fit(X_train_balanced, y_train_balanced)
4
5 rf.score(X_test, y_test)
```

0.734

- All of the classifiers above don't seem to be able to “beat” the RBF SVM classifier
- We think we might find the best solutions using **Deep Learning Algorithms**
 - Neural Networks
- Since “Neural Network” vague term with a lot of parameters we decide to shrink our search space
 - Fully Connected Neural Networks
 - Series of fully connected layers
 - A fully connected layer is a function from $\mathbb{R}^m \rightarrow \mathbb{R}^n$
- The parameters the play a significant role in our architecture are:
 - Number of hidden layers → 2 - 5 hidden layers
 - Number of neurons in every hidden layer → 128, 400, 500, 1024, 2048
 - Non-linear activation function → Tanh, ReLU and LeakyReLU functions
 - We also used dropout layers with $p = 0.4$
 - Tried different values of learning rate and momentum on Stochastic Gradient Descent Optimizer
 - Learning Rate (10^{-3} , 10^{-4} , 10^{-5}) Momentum (0, 0.9)
 - Cross-Entropy Loss function

Part D

Final Neural Network Model (Experiment to find best values of NN)

- We need computational power to this search
- We used the “Aristotelis” cluster (<https://hpc.auth.gr/>)
 - A medium sized institutional HPC infrastructure
 - GPU Partition
 - Used a batch script to submit the job
- Due to the big number of variable parameters, we submitted different versions of the batch script
 - The changes on the script were made manually
- For the training process, we didn't use the (dimension) reduced dataset (by LASSO)
 - We let the NN choose the importance of each feature
 - We trained for a lot of epochs (2000)



Job Results / Best parameters found

4 hidden layers

400 units in every layer

LeakyReLU activation function with negative slope (0.1)

Dropout layers ($p = 0.4$)

Learning Rate = 10^{-5} & Momentum = 0.9 (to avoid oscillations in the loss)

Optimum Batch Size = 1

Part D

Final Neural Network Model Showcase (1)

A function that in a single training loop, the model makes predictions on the training dataset (fed to it in batches), and backpropagates the prediction error to adjust the model's parameters

```
1 # Loss and optimizer initialization
2 optimizer = optim.SGD(params=net.parameters(), lr=1e-5, momentum=0.9)
3 loss_fn = nn.CrossEntropyLoss()
```

To train a NN, we need a loss function and an optimizer

```
1 # Dataset preparation
2 tensor_X_train = torch.tensor(X_train, dtype=torch.float32)
3 tensor_y_train = torch.tensor(y_train, dtype=torch.long)
4 tensor_y_train -= 1
5
6 tensor_X_test = torch.tensor(X_test, dtype=torch.float32)
7 tensor_y_test = torch.tensor(y_test, dtype=torch.long)
8 tensor_y_test -= 1
9
10 myTensorTrainDataset = DataLoader(TensorDataset(tensor_X_train, tensor_y_train), shuffle=True)
11 myTensorTestDataset = DataLoader(TensorDataset(tensor_X_test, tensor_y_test), shuffle=True)
```

This wraps an iterable over our dataset, and supports automatic batching, sampling, shuffling and multiprocess data loading. Default is batch size of 1, so each element in the dataloader iterable will return a batch of 1 random feature and label.

We also check the model's performance against the test dataset to ensure it is learning

```
1 # Train and test loop functions
2 def train_loop(dataloader, model, loss_fn, optimizer):
3     size = len(dataloader.dataset)
4     # Set the model to training mode - important for batch normalization and dropout layers
5     # Unnecessary in this situation but added for best practices
6     model.train()
7
8     # keep and return training loss
9     training_loss = 0
10    for batch, (X, y) in enumerate(dataloader):
11        X = X.to(device)
12        y = y.to(device)
13        # Compute prediction and loss
14        pred = model(X)
15        loss = loss_fn(pred, y)
16
17        # Backpropagation
18        loss.backward()
19        optimizer.step()
20        optimizer.zero_grad()
21        training_loss = loss.item()
22
23    return training_loss
```

```
1 def test_loop(dataloader, model, loss_fn):
2     # Set the model to evaluation mode - important for batch normalization and dropout layers
3     # Unnecessary in this situation but added for best practices
4     model.eval()
5     size = len(dataloader.dataset)
6     num_batches = len(dataloader)
7     test_loss, correct = 0, 0
8
9     # Evaluating the model with torch.no_grad() ensures that no gradients are computed during test mode
10    # also serves to reduce unnecessary gradient computations and memory usage for tensors with requires_grad=True
11    with torch.no_grad():
12        for X, y in dataloader:
13            X = X.to(device)
14            y = y.to(device)
15            pred = model(X)
16            test_loss += loss_fn(pred, y).item()
17            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
18
19    test_loss /= num_batches
20    correct /= size
21
22    return (test_loss, correct)
```

Part D

Final Neural Network Model Showcase (2)

Input is 400 features

```
1 # Network architecture
2 net = nn.Sequential(
3     nn.Dropout(0.4),
4     nn.Linear(400, 400),
5     nn.LeakyReLU(0.1),
6
7     nn.Dropout(0.4),
8     nn.Linear(400, 400),
9     nn.LeakyReLU(0.1),
10
11    nn.Dropout(0.4),
12    nn.Linear(400, 400),
13    nn.LeakyReLU(0.1),
14
15    nn.Dropout(0.4),
16    nn.Linear(400, 400),
17    nn.LeakyReLU(0.1),
18
19    nn.Dropout(0.4),
20    nn.Linear(400, 400),
21    nn.LeakyReLU(0.1),
22
23    nn.Dropout(0.4),
24    nn.Linear(400, 5),
25 )
```

Output is 5 labels' prediction probabilities

We define the layers of the network in the `nn.Sequential` function and specify how data will pass through the network. Modules are added to it in the order they are passed

```
1 epochs = 800
2 train_loss = np.zeros((epochs, 1))
3 test_loss = np.zeros((epochs, 1))
4 test_accuracy = np.zeros((epochs, 1))
5
6 for i in range(epochs):
7     print(f"[*] Epoch {i+1}")
8     train_loss_i = train_loop(myTensorTrainDataset, net, loss_fn, optimizer)
9     (test_loss_i, test_accuracy_i) = test_loop(myTensorTestDataset, net, loss_fn)
10    train_loss[i] = train_loss_i
11    test_loss[i] = test_loss_i
12    test_accuracy[i] = test_accuracy_i
```

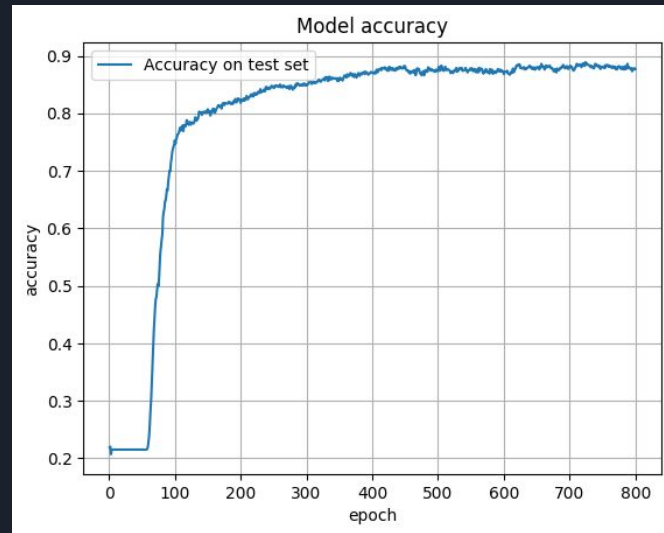
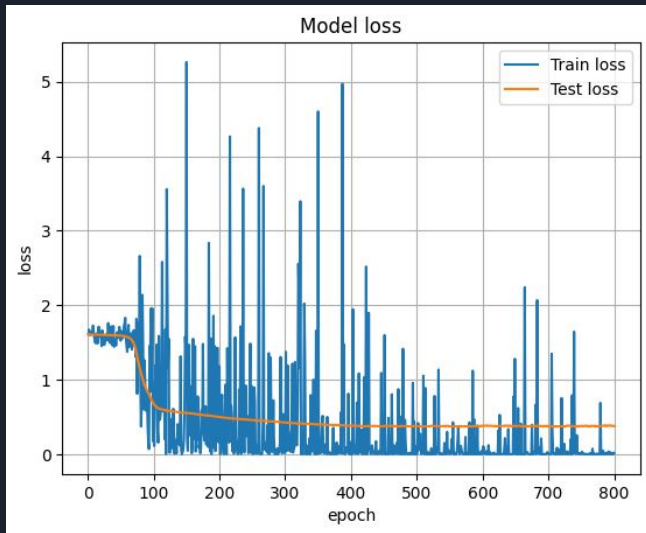
The training process is conducted over several iterations (epochs) During each epoch, the model learns parameters to make better predictions.

- We get the NN's accuracy & loss at each epoch and plot them
 - We'd like to see the accuracy increase with every epoch
 - We'd like the loss decrease with every epoch

Part D

Final Neural Network Model (Results)

We saw convergence in ~800 epochs. So, we re-trained a model for 800 epochs and got:



- The test loss appears to decrease along with the train loss as the epochs pass
 - No overfitting
- Very promising results and our best model yet!!
 - ~89% maximum accuracy reached
 - ~88% accuracy on our final trained model

```
1 print(max(test_accuracy))
2 print(test_accuracy[-1])
```

[0.889]
[0.877]

Part D

Final Neural Network Model (Apply the model to a test set)

X_finalTest

	0	1	2	3	4	5	6	7	8	9	...	390	391	392	393	394	395	396	397	398	399
0	-2.47790	0.68857	0.287380	-0.17280	-1.34050	-0.991000	0.059057	1.106200	0.659300	-0.39487	...	1.485900	0.42348	1.39440	-0.92174	-0.860860	-0.695610	1.016700	0.090674	-1.27330	0.402280
1	-0.20541	0.17982	-1.793900	1.45690	0.25691	-0.239450	0.219400	0.363210	-1.209400	0.53497	...	-2.127000	0.21143	-1.12520	0.50403	0.540360	0.024167	1.093300	1.703100	-0.50955	-0.671590
2	-0.99794	-0.38505	-2.395000	-0.51227	0.16132	0.895100	1.307500	-0.656420	0.928020	-0.91786	...	0.151720	-0.51448	0.11457	-0.61969	-1.478800	-0.035980	0.550380	-0.028087	1.29920	-0.314200
3	1.53080	0.88543	-0.498820	-1.07910	0.13616	0.086067	0.030846	0.211630	0.235680	-1.33610	...	-0.860320	-0.03442	-0.99080	-0.52351	1.117500	0.005782	-0.729210	-1.159300	-0.18932	-1.245500
4	0.54730	-0.39412	-0.529290	-0.04556	-1.39110	0.442700	-0.044157	-2.103300	-0.178400	0.30902	...	-1.189700	0.25043	0.72088	0.37806	0.045692	1.370800	-0.350280	-0.513360	-0.50439	0.104650
...
995	1.08910	0.21326	0.835030	-0.28417	0.27307	-0.056314	-1.010900	0.043052	0.725640	1.28370	...	0.014324	0.25837	-0.48153	0.66335	0.175260	-0.154830	0.050111	0.714430	0.23764	1.347100
996	-0.59062	0.26715	0.347110	-1.33250	0.56694	-0.805930	-1.934900	-0.154540	1.513700	-0.60100	...	-1.004700	-0.66115	-0.36408	0.24614	1.389900	-0.117560	-0.121790	1.787100	-0.21640	-0.788620
997	0.26128	0.88843	-0.040095	-0.24175	-0.52194	-0.405620	1.135000	0.334720	-0.078696	1.71100	...	-0.103000	-0.12294	1.23640	-1.05740	0.361620	-1.782500	0.589480	0.429690	-1.08450	-0.792160
998	0.63851	-0.37042	-0.847930	-0.67745	0.20323	1.400100	0.396640	-0.687600	1.386500	1.37040	...	1.134600	-0.75447	-0.90325	0.95127	0.462900	-0.503690	-0.668310	0.120380	-1.28320	-0.430640
999	0.12987	1.28310	-1.005600	0.77731	1.55160	0.314390	-1.311200	-1.418100	-0.679630	-0.47039	...	0.205680	0.69881	-1.13290	0.29076	0.307100	1.796800	1.223200	-0.310640	1.29130	-0.092431

1000 rows x 400 columns

datasetCTest.csv

Final Trained
Full Connected NN model

- After saving the file, we did some validations to ensure that file was saved correctly and can be read using `numpy.load()`

```
1 print(labels42)

[5 2 4 1 4 1 2 4 3 4 5 1 2 1 5 5 3 5 1 4 5 5 1 1 2 1 3 2 2 2 2 1 2 2 3
 5 3 1 3 2 4 4 2 1 5 2 4 3 2 1 2 1 3 2 1 2 4 1 1 3 2 1 3 3 1 3 4 1 4 3 1
 1 2 4 3 5 1 5 4 4 1 5 5 4 5 2 4 2 3 3 2 4 3 2 4 3 4 1 1 4 5 5 2 4 3 2 3
 5 1 4 5 3 1 5 3 4 2 1 1 1 4 5 3 2 5 2 1 5 5 1 4 1 2 1 4 2 4 4 2 4 1 5 2 1
 4 2 3 1 2 3 3 3 4 5 4 1 1 1 2 3 4 4 5 2 3 1 1 1 2 2 3 3 5 4 5 1 1 5 5 3 3 4
 2 4 5 1 3 5 3 2 1 3 2 1 2 5 1 3 1 2 1 3 3 3 2 5 4 5 2 4 5 1 1 5 3 3 1 1
 4 4 4 2 5 4 4 1 3 2 1 2 3 4 4 3 3 4 2 5 1 2 4 5 1 3 3 1 1 3 4 4 2 3 2 2
 5 1 5 1 5 1 5 4 2 1 1 4 4 1 1 5 2 1 3 4 4 2 5 1 1 2 2 3 5 4 5 4 5 1 1 2 3
 3 4 3 5 4 1 1 4 2 4 1 1 2 1 4 4 3 4 4 4 4 4 5 3 5 3 2 1 5 1 5 3 1 3
 1 4 2 1 1 3 3 3 4 2 5 1 1 2 2 4 1 4 4 3 2 5 5 4 5 1 1 2 1 2 4 3 5 3 2 4 2
 5 1 5 3 1 5 4 3 4 3 3 5 3 2 4 5 3 2 5 4 2 5 1 2 3 4 5 3 5 2 1 5 2 3 4 5
 4 5 5 1 1 2 3 3 4 3 5 4 4 1 1 1 4 2 3 4 2 4 4 1 3 1 4 4 5 3 3 5 1 3 3 4 1 1
 3 4 5 1 5 3 1 2 4 3 3 5 1 4 4 1 4 2 3 1 4 1 4 2 1 4 5 2 4 5 2 1 2 4 3 5
 2 5 2 4 5 2 4 3 4 1 2 2 2 1 3 1 5 2 4 3 5 5 4 3 2 2 1 4 2 4 4 3 3 3 2 4 2
 1 3 4 4 5 3 2 2 3 3 3 4 3 1 4 4 1 5 1 3 5 1 1 5 2 1 4 3 1 2 1 3 2 2 1
 5 2 3 2 2 1 2 2 3 1 4 3 5 5 4 4 4 1 1 5 5 1 4 4 2 1 4 4 5 3 1 1 2 4 4
 3 3 1 5 4 2 3 3 1 3 2 1 4 1 3 3 3 4 1 1 2 3 3 2 1 5 5 1 2 3 1 1 2 4 4
 2 3 5 1 4 5 1 5 2 1 1 1 1 5 3 2 1 1 2 3 1 2 2 3 4 3 3 1 5 1 4 1 2 3 4 5
 5 1 1 3 2 3 1 4 1 4 2 1 5 1 3 1 1 2 4 1 5 2 4 2 4 1 4 3 5 2 1 5 3 5 1 1
 4 3 1 4 5 1 4 2 4 1 1 4 2 1 1 3 2 1 2 4 2 3 5 2 4 4 5 3 5 4 4 5 2 3 5 3
 3 5 5 2 5 2 4 4 4 5 1 2 1 5 5 1 1 3 4 2 1 2 3 2 1 5 3 2 1 3 1 3 3 1 3 1
 4 1 2 3 3 1 2 1 2 1 4 1 2 5 1 2 2 4 1 5 5 1 5 5 2 1 1 1 1 5 3 1 2 1 4
 4 4 5 5 1 1 5 4 1 2 2 4 2 4 3 4 4 2 5 4 5 1 3 1 2 4 5 1 1 1 4 4 4 2 5
 5 2 4 2 5 5 3 1 3 4 4 3 1 2 1 2 2 2 1 1 2 1 5 4 1 4 2 2 3 3 3 1 4 1 1 4 2
 2 5 1 3 3 4 5 3 5 1 2 3 4 5 5 3 1 2 2 3 4 5 4 4 2 1 3 5 2 5 2 5 2 1 2 1
 4 3 5 2 1 1 2 5 3 5 2 2 1 4 3 4 1 2 2 1 3 4 4 1 3 5 5 3 4 5 5 1 1 1 3 1 1
 4 4 3 3 2 4 5 4 5 1 1 2 1 2 3 2 5 4 1 5 4 4 1 2 3 4 1 5 1 2 5 1 4 5 3 5 4
 1]
```

```
1 np.save('/content/pattern_recognition_assignment/labels42.npy', labels42)
```



labels42.npy

Thank You!

For Your Attention

Any questions?



https://github.com/Kyparissis/pattern_recognition_assignment