PHYS555 - Machine Learning Project

1 - Describe how SVM algorithms can be used for classification and regression problems (describe the algorithms). Which parameters are the most important ones in the models for classification and regression (e.g., for fitting and controlling overfitting...)? What is the difference between classification and regression algorithms in SVM?

2- Use 'sklearn.svm.SVC' to classify your data. Then, compare results (taken from different metrics such as confusion matrices, accuracy, recall and precision, recall and AUC) with the best results obtained from an ANN model and discuss the comparisons.

3- Use 'sklearn.svm.SVR' for regression. Compare results (using regression metrics such as scatter, mean, median and associated plots/histograms) with the best results from an ANN model and discuss the comparisons.

Karlee Zammit - V00823093

Introduction

In this notebook, I will discuss what a support vector machine (SVM) is, how the algorithm works, parameters to avoid overfitting, and the difference between the classification and regression SVM algorithms. I then use an artificial neural network and SVM for classification and regression tasks, and discuss the performance of both.

Support Vector Machine

A SVM is a supervised learning model (or decision machine) that analyzes data for classification and regression tasks. At a high level, the SVM algorithm maximizes a particular mathematical function with respect to a given collection of data. As discussed in Bishop, C. M. (2006)., an important property of SVM is that any local solution is also a global optimum.

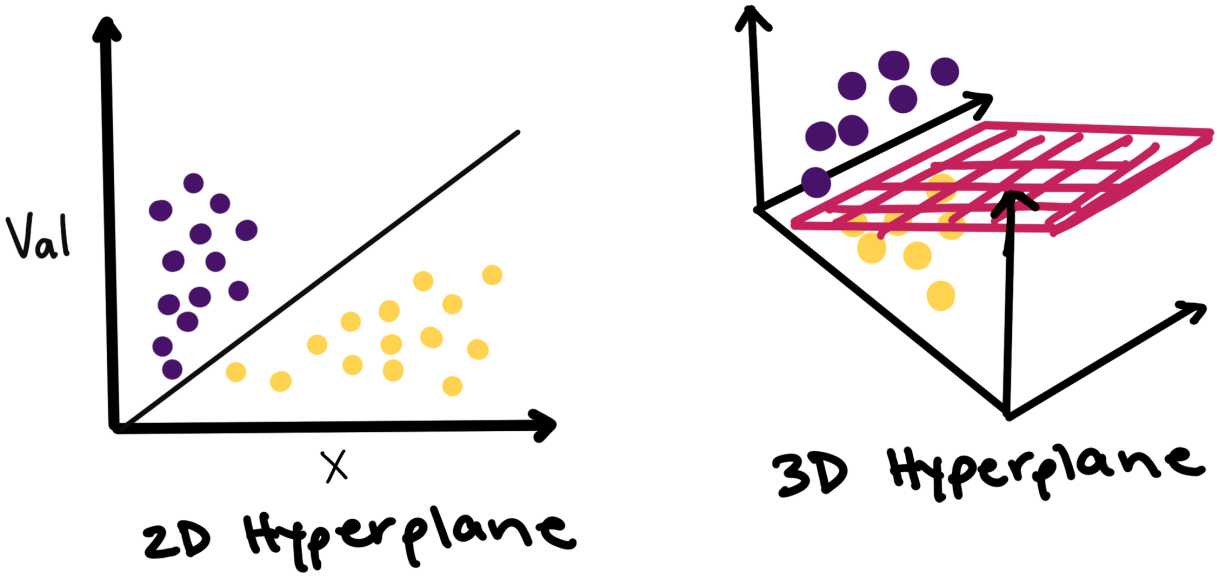
Before diving into the math behind the algorithm and the algorithm itself, I present four key concepts as discussed in Noble, W. (2006):

1. The separating hyperplane
2. The maximum-margin hyperplane
3. The soft margin
4. The kernel function

I will discuss these below for a binary classification example with linear SVM. Binary classification with SVM can be extended to multiclass classification with a "one vs rest" approach.

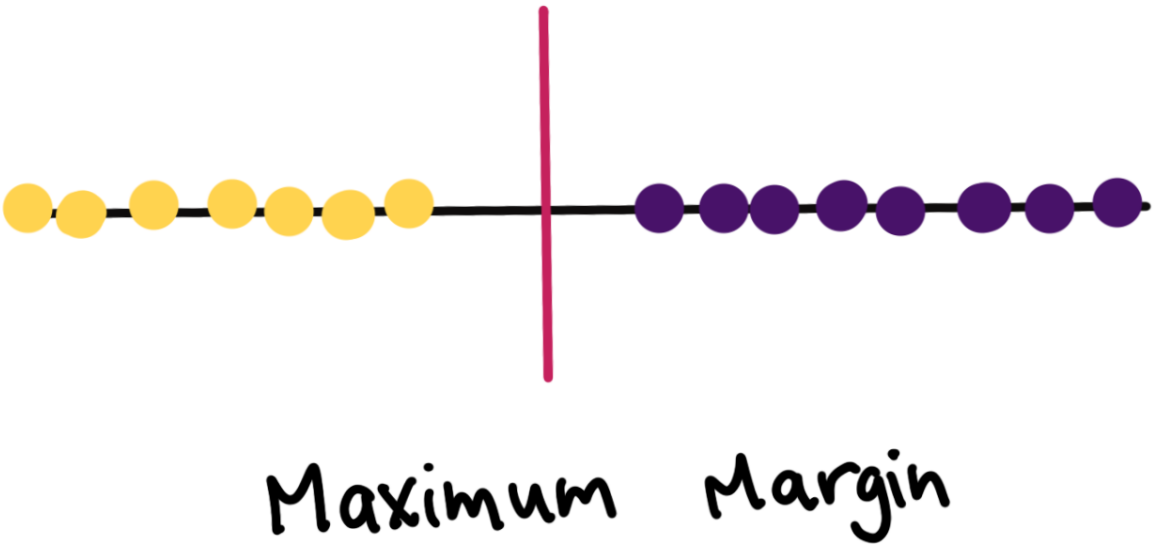
The Separating Hyperplane

For an imaginary dataset, that looks like the left panel of the Figure below (titled "2D Hyperplane"), a separating line can be drawn through the data. Then for a future prediction, depending on where the prediction falls on the graph, a classification can be made if it will belong to the purple or orange class. This separating line is called the separating hyperplane. This idea can be extended to higher dimensions, with a 3-dimensional example provided in the right panel of the Figure below (titled "3D Hyperplane).



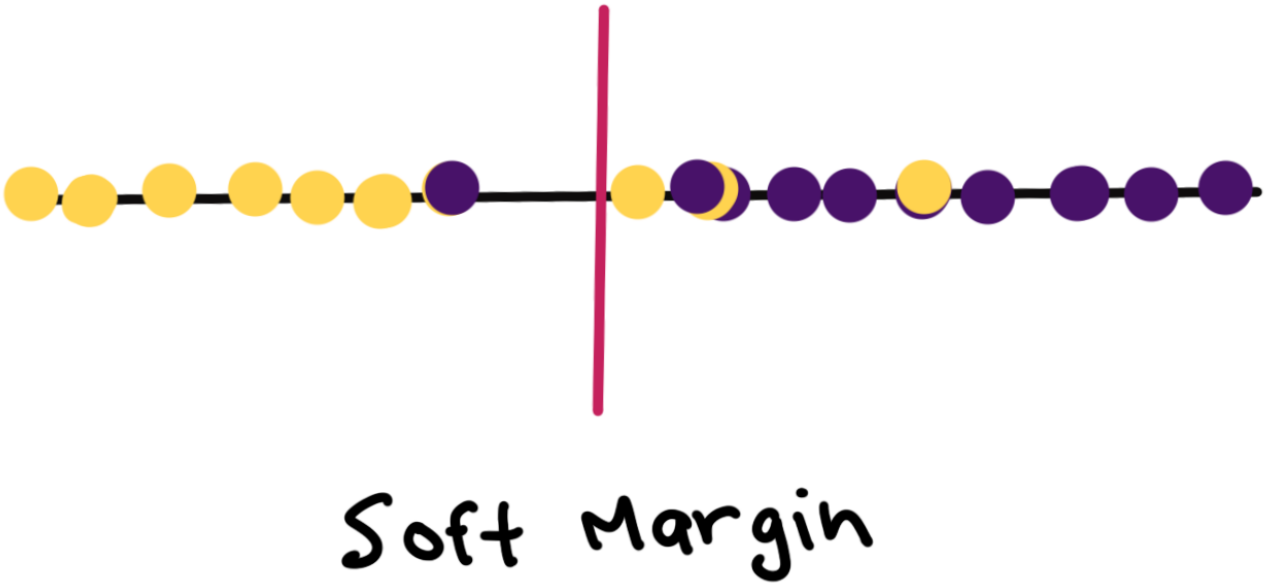
The Maximum-Margin Hyperplane

In a 1D example as shown in the Figure below, the "maximum-margin" hyperplane is located at the position in space that maximizes it's distance from each of the two classes. If you were to move this margin closer to one class, it would no longer be the maximum distance away and therefore would have a higher chance of inaccurately predicting a future observation of each class. For linearly separable like in the Figure below, the maximum-margin hyperplane can be used to determine the optimized hyperplane location.



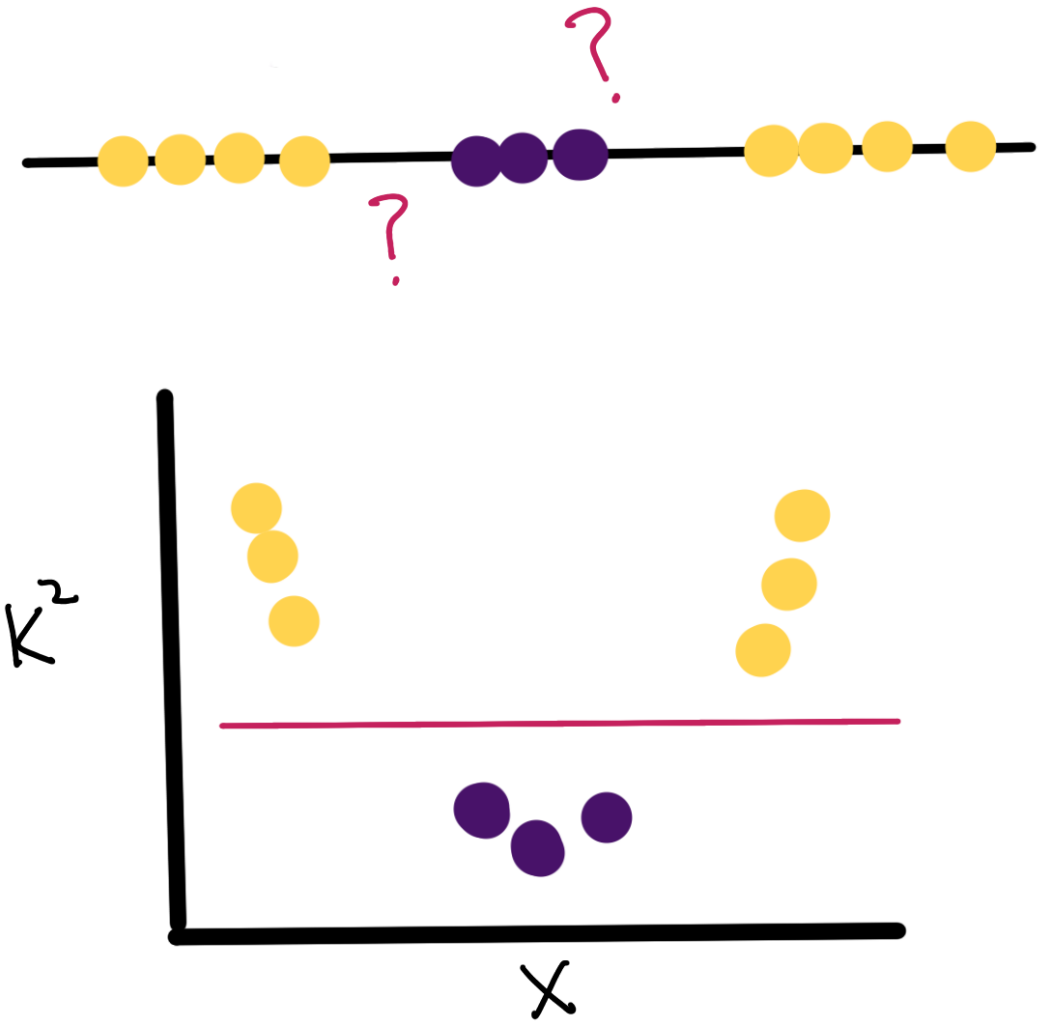
But what if the data was not easily linearly separatable, as shown in the Figure below? It would then be ideal to allow for misclassifications, so that future observations can be more accurately predicted (ie. avoid overfitting to the data). This is an example of the tradeoff between bias and variance, which is a common

theme in machine learning algorithms. The location of this soft margin is determined by trial and error using cross validation.



The Kernel Function

Sometimes data is too complex to be overcome by the introduction of a soft margin alone. For example, in the top panel of the Figure below, there exists no linear solution that could separate the two classes from one another. The kernel function provides a solution to the problem, adding an additional dimension to the data. In this example, by squaring the original values, a new dimension is introduced and a linear solution can then be used to separate the classes from one another. It can be proven that for any given labelled data set, there exists a kernel function that allows the data to be linearly separated. One needs to consider the curse of dimensionality here, as complex data can be projected into higher and higher dimensions, increasing computational requirements.

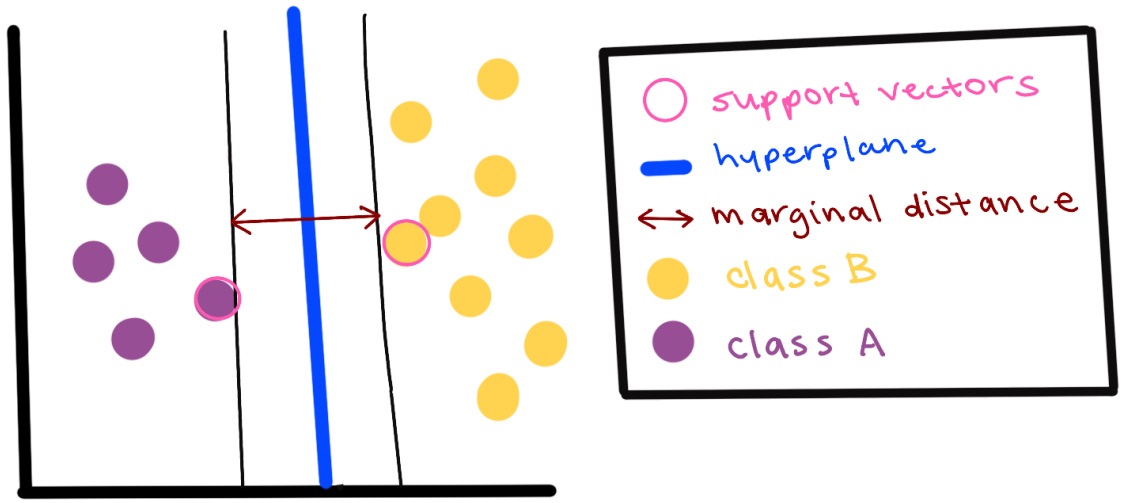


The Algorithm

For any data set, there may be multiple hyperplanes that exist that can separate the data. Support vectors are data points that are close to the hyperplane and influence the position and orientation of the hyperplane. A variety of parameters are important in fine-tuning the optimized hyperplane, and these are discussed below in the "Avoiding Overfitting" section.

For classification, SVM finds the optimal hyperplane solution by maximizing the distance between the two classes (allowing for a soft boundary when necessary).

For regression, SVM finds the optimal hyperplane, allowing for data points at a distance epsilon away from the hyperplane to be included with no increased error.

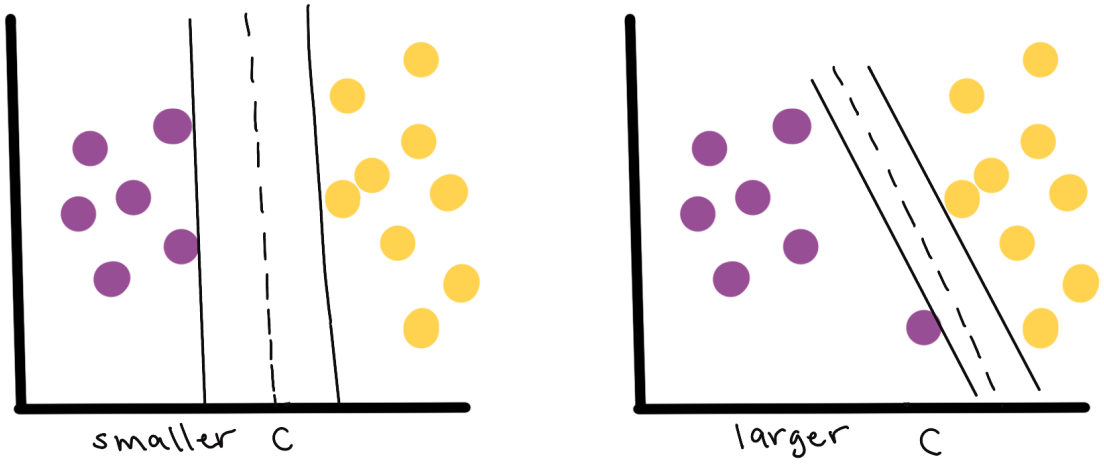


Avoiding Overfitting

For SVC, the most important parameters for avoiding overfitting are "C" and "Gamma". For SVR, the most important parameters are "C", "Gamma", and "Epsilon".

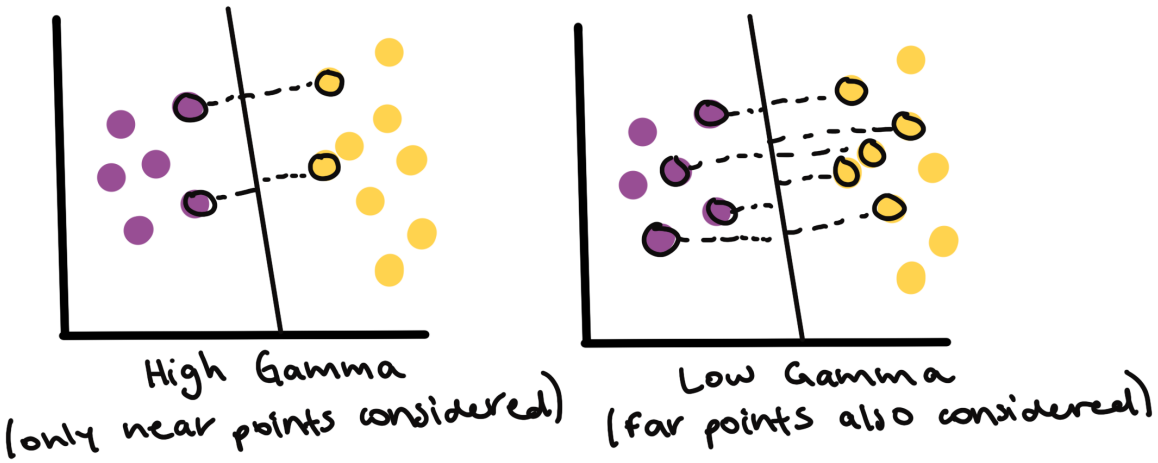
C (Regularization Parameter)

C adds a penalty for each misclassified data point, meaning it tells the SVM optimization how much you want to avoid misclassifying each training example. If C is small, there is a small penalty for misclassified points, and so the decision boundary with a large margin is chosen at the expense of many misclassified points. If C is large, SVM tries to minimize the number of misclassified examples, which results in a decision boundary with a smaller margin. If C is too large, this can cause overfitting.



Gamma (Kernel Coefficient)

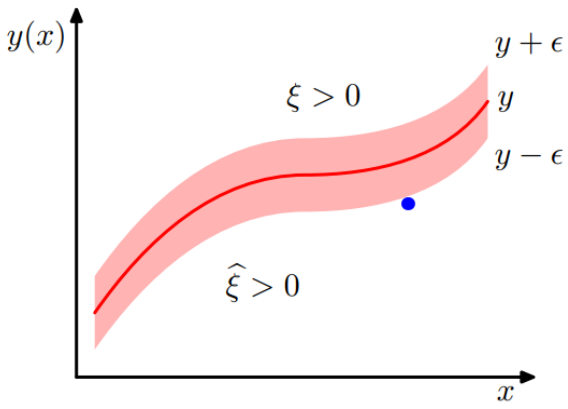
Gamma controls the distance of the influence of a single training point. Low values of gamma result in a large similarity radius, and so more points are grouped together. High values of gamma mean that less points need to be grouped together in order to be considered in the same group or class. Large gamma values tend to lead to overfitting.



Epsilon (for Regression Only)

The below figure from Chapter 7 of Bishop, C. M. (2006). provides an excellent summary of epsilon.

Figure 7.7 Illustration of SVM regression, showing the regression curve together with the ϵ-insensitive 'tube'. Also shown are examples of the slack variables ξ and $\widehat{\xi}$. Points above the ϵ-tube have $\xi > 0$ and $\widehat{\xi} = 0$, points below the ϵ-tube have $\xi = 0$ and $\widehat{\xi} > 0$, and points inside the ϵ-tube have $\xi = \widehat{\xi} = 0$.



The error for points within the epsilon away from the optimal hyperplane is disregarded. Another name for this is called the "epsilon insensitive tube".

Math Behind the Algorithm (Bishop, C. M. (2006) Summary)

Chapter 7 of Bishop, C. M. (2006) provides an excellent explanation of the SVM algorithm, of which I provide a brief summary below.

For a linear function of the form

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) + b$$

where w are the polynomial coefficients, φ is a fixed feature-space transformation, and b is an explicit bias.

SVM solves the optimization problem:

$$\arg\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

through the use of Lagrange multipliers.

We introduce a slack variable ξ, which allows data points to be on the "wrong side" of the margin boundary, with a penalty that increases with distance from that boundary.

For **classification**, to maximize the margin while applying a penalty to points that lie on the wrong side of the margin, we minimize

$$C\sum_{n=1}^{N}\xi_n + \frac{1}{2}\|\mathbf{w}\|^2$$

where C>0, and controls the trade-off between the slack variable penalty and the margin.

For **regression**, using Langrage multipliers, the error function

$$C\sum_{n=1}^{N}(\xi_n + \widehat{\xi}_n) + \frac{1}{2}\|\mathbf{w}\|^2$$

is minimized, where ξ-hat is introduced by the epsilon-tube described in the above section.

Summary:

The two goals of SVM for classification are:

- Increase the distance of decision boundary to classes (or support vectors)
- Maximize the number of points that are correctly classified in the training set

The goal of SVM for regression is to determine an optimal hyperplane and epsilon-tube containing the maximum number of points from the data.

Classification

In this section, I will compare the performance of an ANN (artificial neural network) and SVM (discussed in previous question), with the target of predicting what type of forest covers a specific area. I will be using the "Covertype" data set, obtained from the UCI machine learning repository. The "Covertype" data set has 54

attributes and 581012 instances. There are 7 different classes within this dataset. It has been used in multiple papers (links available on the UCI website, linked in the references below) to compare the performance of multiple ANNs.

This dataset is imbalanced, meaning that there are more samples of class 1 and 2 than the other classes. To balance the data, I use undersampling, chosen instead of other resampling methods due to the time constraints of this project and computational power available. I then run both the ANN and SVM multiple times, taking the average of the results, and presenting error as a standard deviation. This is done so that, even though I have randomly undersampled the data, the results can be generalized for the entire data set and future classifications.

This section is laid out as follows:

1. Load and present the data
2. Data visualization
3. ANN and SVM runs (optimized previously with pipelines)
4. Results
5. Conclusion, comparing the results

In [1]:
```
# Import the necessary packages
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.svm import SVC
from imblearn.under_sampling import CondensedNearestNeighbour
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import itertools
from sklearn.neural_network import MLPClassifier
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_curve, auc
from itertools import cycle
from scipy import interp
import scikitplot as skplt
from sklearn.multiclass import OneVsRestClassifier
import seaborn as sns
from tabulate import tabulate
from sklearn.svm import SVR
from collections import Counter
```

Functions

In [31]:
```
# Confusion matrix function
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.viridis):

    #cmap=plt.cm.Blues
```

```
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    #print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()


def plot_ROC_curve(va_prob, Y_va, title):
# Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(7):
        fpr[i], tpr[i], _ = roc_curve(Y_va[:, i], va_prob[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(Y_va.ravel(), va_prob.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Plot ROC curve
    plt.figure()
    plt.plot(fpr["micro"], tpr["micro"],
             label='micro-average ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["micro"]))
    for i in range(7):
        plt.plot(fpr[i], tpr[i], label='ROC curve of class {0} (area = {1:0.2f})'
                                       ''.format(i, roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(str(title))
    plt.legend(loc="lower right")
    plt.show()

def PCA_analysis(df):

    # Initialize the normalization estimator
```

```
    sc = StandardScaler()

    # Train the estimator on the input data. This method calculates the mean and varianc
    sc.fit(df)

    # Apply the method to the a data, to transform all of the features using their respe
    df_norm = sc.transform(df)

    # Initialize scikit learns principal component analysis function
    pca = PCA()

    # Determine transformed features
    df_analysis_pca = pca.fit_transform(df_norm)

    # Determine explained variance using explained_variance_ration_ attribute
    exp_var_pca = pca.explained_variance_ratio_

    # Cumulative sum of eigenvalues; This will be used to create step plot
    # for visualizing the variance explained by each principal component.
    cum_sum_eigenvalues = np.cumsum(exp_var_pca)

    # Create the visualization plot
    plt.bar(range(0,len(exp_var_pca)), exp_var_pca, alpha=0.5, align='center', label='In
    plt.step(range(0,len(cum_sum_eigenvalues)), cum_sum_eigenvalues, where='mid',label='
    plt.ylabel('Explained Variance Ratio')
    plt.xlabel('Principal Component Index')
    plt.title('Explained Variance Ratio vs. Principal Component Index')
    plt.legend(loc='best')
    plt.tight_layout()
    plt.show()
```

Load the data

In [3]:
```
# Load in data and print out a few rows, as well as the shape of the data
df = pd.read_csv('covtype.csv')
print('The shape of the dataset is ' + str(df.shape))

# print the first few columns of the dataframe
df.head()
```

```
The shape of the dataset is (581012, 55)
```

Out[3]:

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance
0	2596	51	3	258	0	
1	2590	56	2	212	-6	
2	2804	139	9	268	65	
3	2785	155	18	242	118	
4	2595	45	2	153	-1	

5 rows × 55 columns

Looking at the above data, there is both binary and continuous data. For practice, I will use OneHot encoding for the cover type variable, even though it is already in numerical format.

In [4]:
```
# Assign "cover type" as the target variable
X = df.drop(["Cover_Type"], axis=1).values
y = df["Cover_Type"].values
```

```
# Print the number of samples within each class in the unaltered data
print('Original dataset shape %s' % Counter(y))
```

```
Original dataset shape Counter({2: 283301, 1: 211840, 3: 35754, 7: 20510, 6: 17367, 5: 9
493, 4: 2747})
```

As you can see, this is an imbalanced data set, where classes 1 and 2 have thousands more samples than the other classes. To avoid this issue (ie. ANN having bias towards predicting one class better than another), I have chosen to resample the data and use this sampled data going forward. There are many different ways to resample data, with some good options being KNN condensed neighbours to determine which samples to keep. For the sake of this assignment, I will use random undersampling as this data set is large and my computing power is limited.

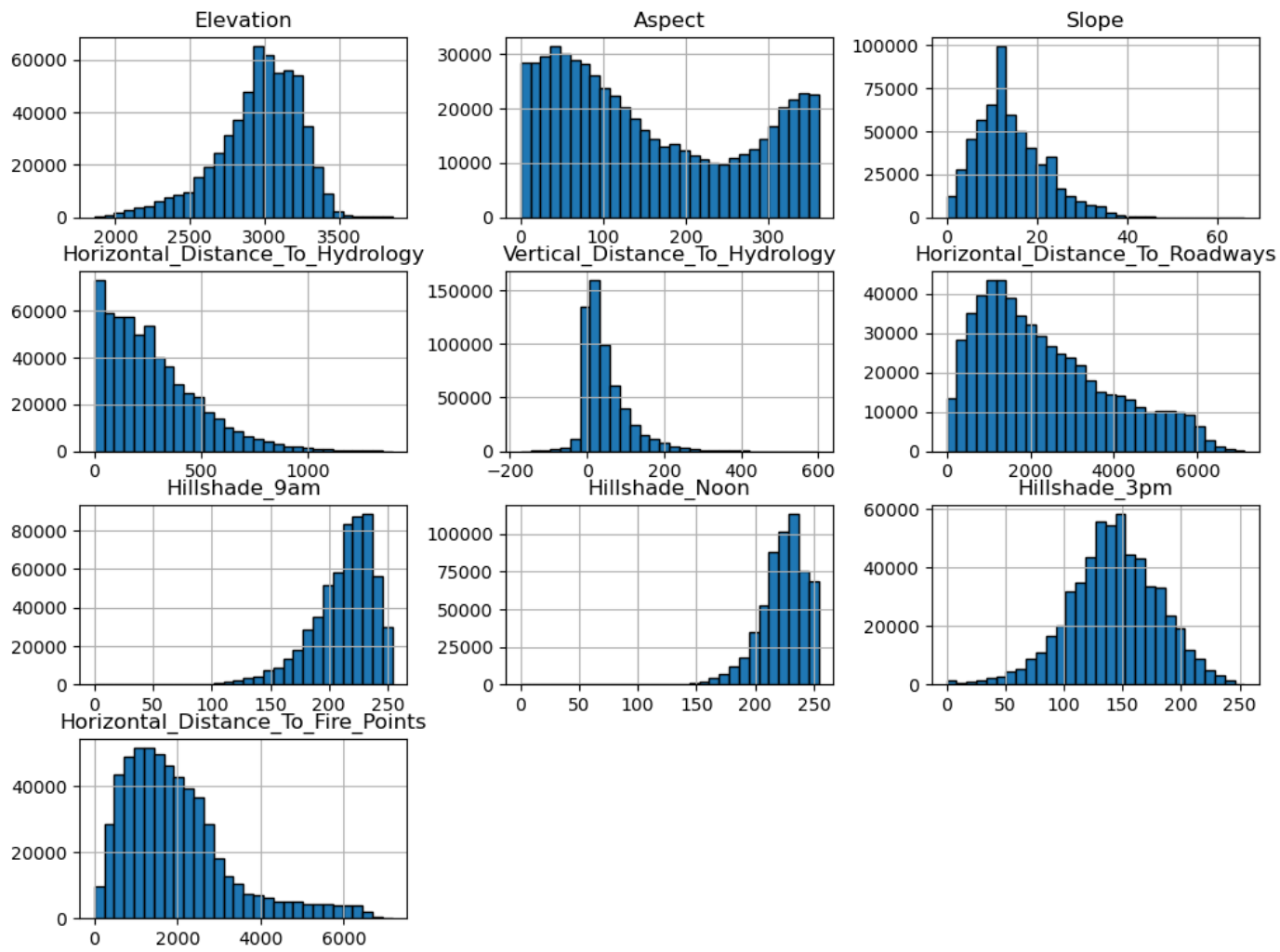
I chose not to apply principal component analysis for this analysis, as more information is always better, and I did not experience significant computational restrictions due to the dimension of the data.

Visualize the data

Below are histograms of each continuous variable within the data set, excluding the target.

In [34]:
```
# Plot a histogram of the continuous variables, and the target variable
df_cont = df.drop(['Wilderness_Area1',
       'Wilderness_Area2', 'Wilderness_Area3', 'Wilderness_Area4',
       'Soil_Type1', 'Soil_Type2', 'Soil_Type3', 'Soil_Type4', 'Soil_Type5',
       'Soil_Type6', 'Soil_Type7', 'Soil_Type8', 'Soil_Type9', 'Soil_Type10',
       'Soil_Type11', 'Soil_Type12', 'Soil_Type13', 'Soil_Type14',
       'Soil_Type15', 'Soil_Type16', 'Soil_Type17', 'Soil_Type18',
       'Soil_Type19', 'Soil_Type20', 'Soil_Type21', 'Soil_Type22',
       'Soil_Type23', 'Soil_Type24', 'Soil_Type25', 'Soil_Type26',
       'Soil_Type27', 'Soil_Type28', 'Soil_Type29', 'Soil_Type30',
       'Soil_Type31', 'Soil_Type32', 'Soil_Type33', 'Soil_Type34',
       'Soil_Type35', 'Soil_Type36', 'Soil_Type37', 'Soil_Type38',
       'Soil_Type39', 'Soil_Type40', 'Cover_Type'], axis=1)

# Plot histogram of all continuous variable
df_cont.hist(figsize=(12, 9), bins=30, edgecolor="black")
```

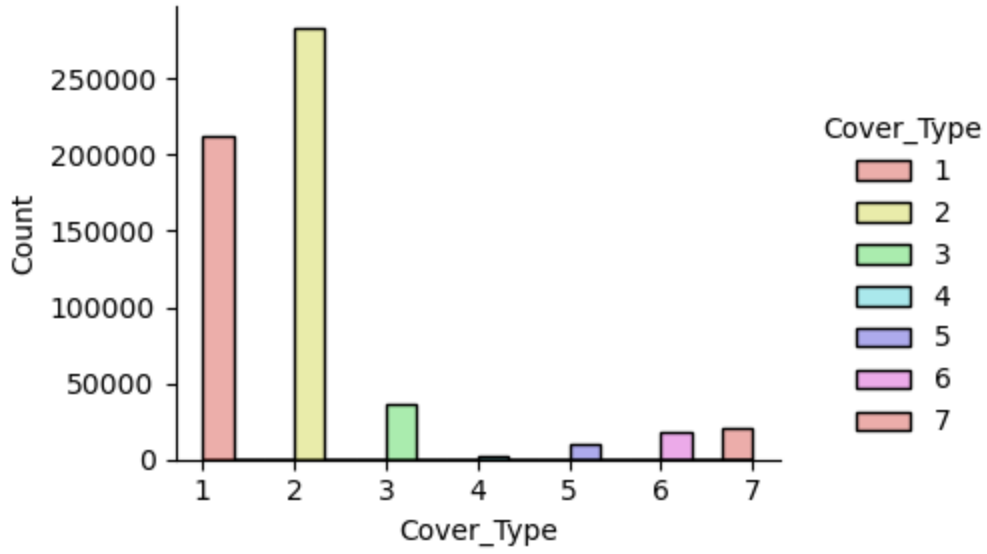
Out[34]:
```
array([[<AxesSubplot:title={'center':'Elevation'}>,
        <AxesSubplot:title={'center':'Aspect'}>,
        <AxesSubplot:title={'center':'Slope'}>],
       [<AxesSubplot:title={'center':'Horizontal_Distance_To_Hydrology'}>,
        <AxesSubplot:title={'center':'Vertical_Distance_To_Hydrology'}>,
        <AxesSubplot:title={'center':'Horizontal_Distance_To_Roadways'}>],
       [<AxesSubplot:title={'center':'Hillshade_9am'}>,
        <AxesSubplot:title={'center':'Hillshade_Noon'}>,
        <AxesSubplot:title={'center':'Hillshade_3pm'}>],
       [<AxesSubplot:title={'center':'Horizontal_Distance_To_Fire_Points'}>,
        <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```



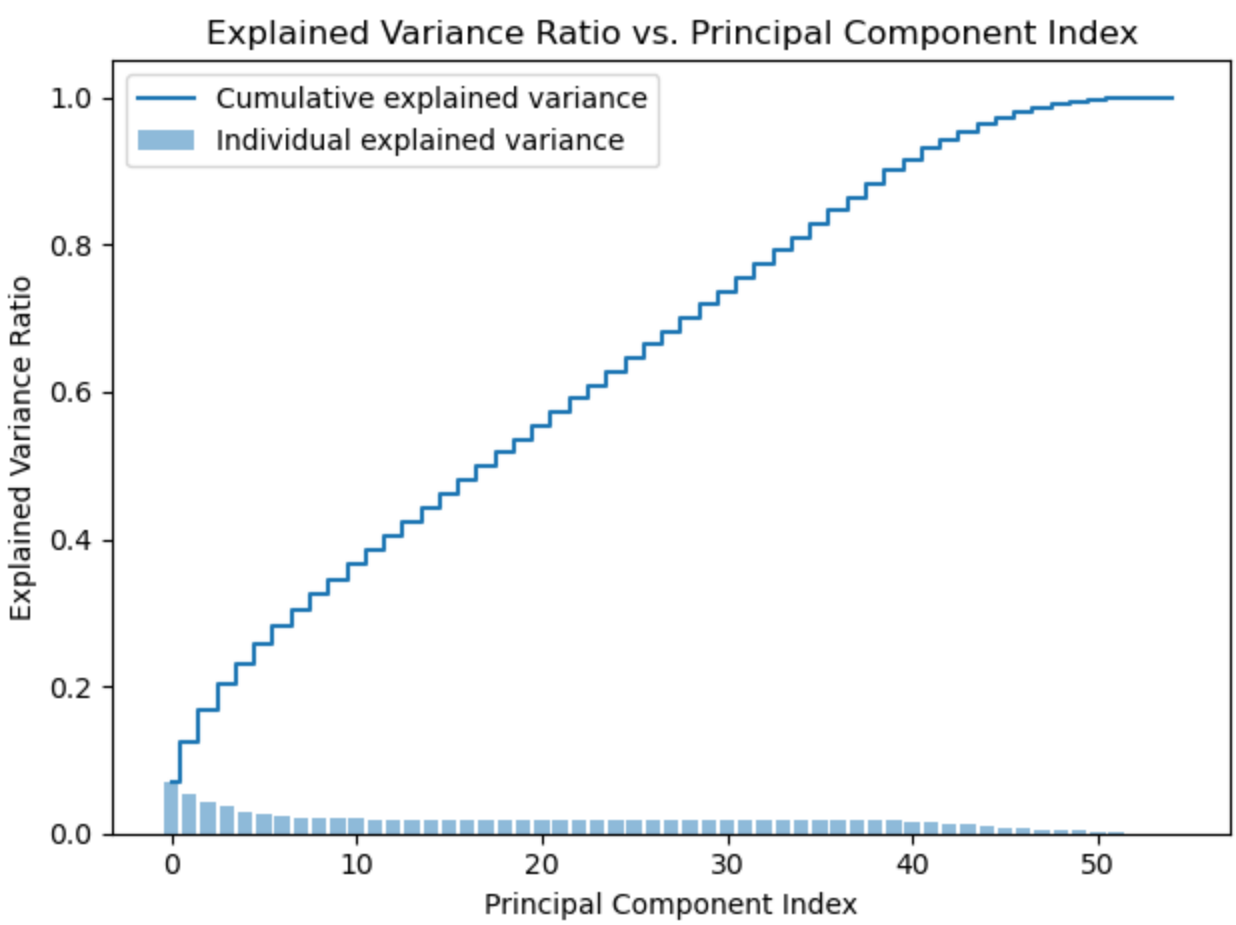
Below is a histogram of the target - note that the classes are imbalanced.

In [33]:
```
# Let's plot the cover type histogram again
sns.displot(data=df, x='Cover_Type', bins=18,
            height=3, aspect=1.4, hue='Cover_Type',
            palette='hls')

# Horizontal_Distance_To_Fire_Points
```

Out[33]:
```
<seaborn.axisgrid.FacetGrid at 0x16210fbe988>
```



In [32]:
```
PCA_analysis(df)
```



As shown in the above PCA analysis plot, more than 45 components are needed to explain the total variance in the data set. So, I will not apply PCA before this analysis to ensure all information is kept for the classification process.

ANN and SVM Runs

Below are the two pipelines that I used to optimize the ANN and SVM algorithms. These are commented out, as I have encorporated these optimized models into the loop below, but wanted to show the optimization process.

In [8]:
```
### ANN PIPELINE ###

# From previous pipelines: activation, solver, validation fraction, learning rate, hidde
# these were the best parameters before I tested alpha and tolerance

#ann_pipeline = Pipeline([('ANNcls', MLPClassifier(early_stopping=True, n_iter_no_change
#                                                  learning_rate='adaptive', solver='sgd
#                                                 hidden_layer_sizes=(20,50,20), activat

#params = [{'ANNcls__alpha':[1, 0.1, 0.01, 0.001], 'ANNcls__tol':[0.0001, 0.001, 0.01, 0

#gs_ann = GridSearchCV(ann_pipeline,
#                      param_grid=params,
#                      scoring='accuracy',
#                      cv=10)

#gs_ann.fit(X_tr_Norm, Y_tr)

#gs_ann.best_params_

### SVM PIPELINE ###
```

```
#\sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=
# class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', break_ti

#svm_pipeline = Pipeline([('PCA', PCA()), ('svc', SVC(decision_function_shape = 'ovr'))]

#params = [{'PCA__n_components':[.7, .9, .999], 'svc__kernel':['linear', 'poly', 'rbf',

#gs_svm = GridSearchCV(svm_pipeline,
#                      param_grid=params,
#                      scoring='accuracy',
#                      cv=5)

#gs_svm.fit(X_tr_Norm, Y_tr.argmax(axis=1))

#gs_svm.best_params_
```

Below is the loop for both the ANN and SVM algorithms. The loop plots:

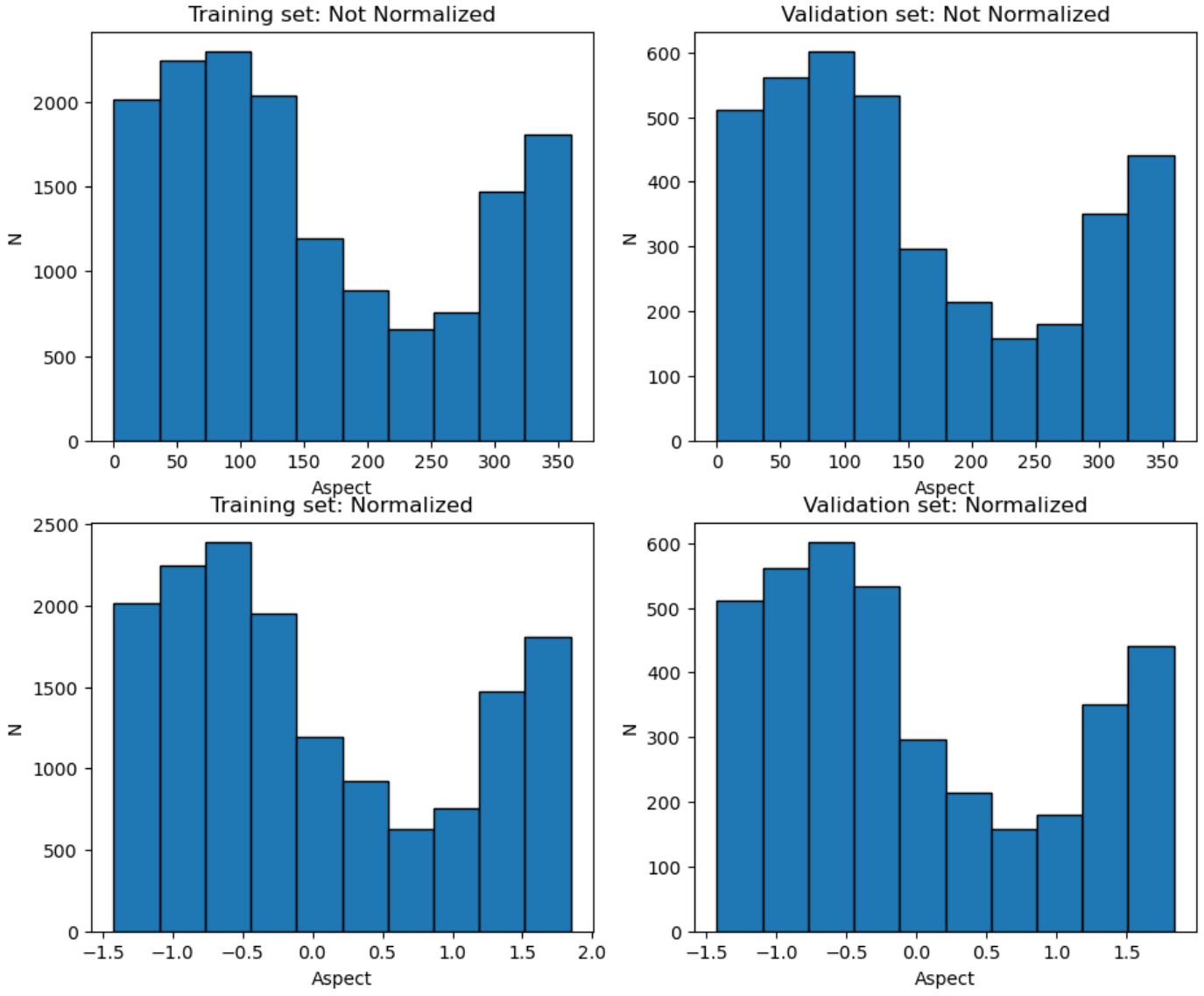
1. On iteration 0: normalized vs. unnormalized histograms for one example variable, balanced cover type histogram
2. On all iterations: ROC curve plot for both the ANN and SVM for that iteration

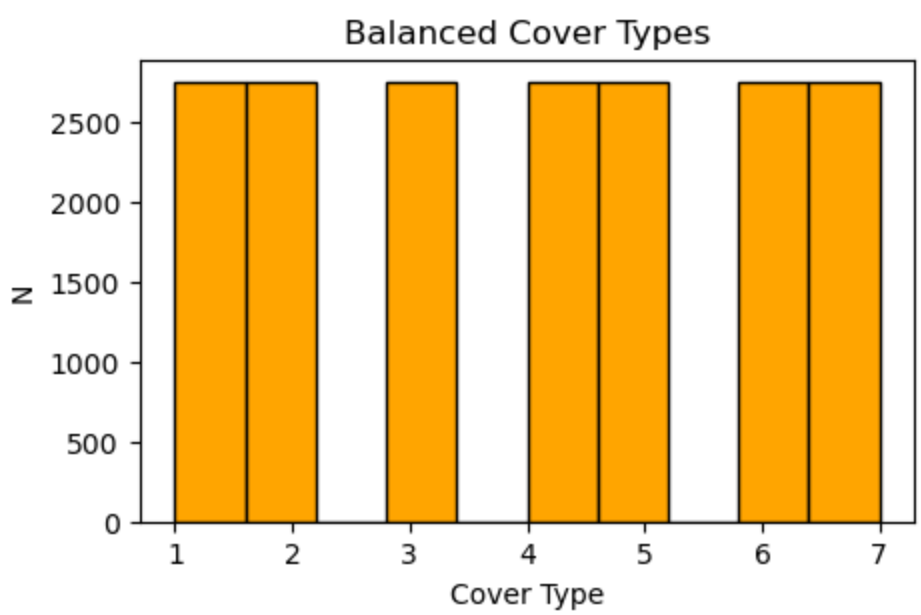
In [9]:
```
# Set the number of iterations to run the ANN and SVM algorithms
n_iter = 5

# Initialize empty variables to keep track of necessary results
class_report_ann = []
cnf_matrix_ANN_tr = []
cnf_matrix_ANN_va = []

class_report_svm = []
cnf_matrix_SVM_tr = []
cnf_matrix_SVM_va = []

# In each iteration,
for ii in range(n_iter):

    print('Starting iteration ' + str(ii+1))

    # undersample the data randomly
    print('Undersampling the data')
    #rus = RandomUnderSampler(random_state=42)
    rus = RandomUnderSampler()
    X_res, y_res = rus.fit_resample(X, y)
    #print('Resampled dataset shape %s' % Counter(y_res))

    # Practice using OneHotEncoder, although this is not necessary for this data set.
    ohe = OneHotEncoder(sparse=False)
    tar= np.reshape(y_res,(-1,1))
    tar = ohe.fit_transform(tar)

    # Split the data into training and validation, with 20% test size
    print('Splitting the data')
    # Split the data into training and validation, with 20% of the data for validation.
    #X_tr, X_va, Y_tr, Y_va = train_test_split(X_res, tar, test_size=0.20)
    X_tr, X_va, Y_tr, Y_va = train_test_split(X_res, tar, test_size=0.20, random_state=4
    #print ('training set == ',np.shape(X_tr),np.shape(Y_tr),',, validation set == ', np

    # Normalize the data
    print('Normalizing the data')
    # Normalize the data using the StandardScaler function, ie. center the mean of the d
    scaler_S= StandardScaler().fit(X_tr)  # line #2
    X_tr_Norm= scaler_S.transform(X_tr) # line # 3
```

```
    X_va_Norm= scaler_S.transform(X_va)   # Line #4

    # Plot an example of the normalized data, and the balanced cover type data
    if ii == 0:
        print('Plot of Normalized vs Not Normalized Data')
        fig = plt.figure(figsize=(11, 9))
        n_col = 1
        title = 'Aspect'
        plt.subplot(2, 2, 1)
        plt.hist(X_tr[:,n_col], edgecolor='black')
        plt.title('Training set: Not Normalized')
        plt.ylabel('N')
        plt.xlabel(str(title))
        plt.subplot(2, 2, 2)
        plt.hist(X_va[:,n_col], edgecolor='black')
        plt.title('Validation set: Not Normalized')
        plt.ylabel('N')
        plt.xlabel(str(title))
        plt.subplot(2, 2, 3)
        plt.hist(X_tr_Norm[:,n_col], edgecolor='black')
        plt.title('Training set: Normalized')
        plt.ylabel('N')
        plt.xlabel(str(title))
        plt.subplot(2, 2, 4)
        plt.hist(X_va_Norm[:,n_col], edgecolor='black')
        plt.title('Validation set: Normalized')
        plt.ylabel('N')
        plt.xlabel(str(title))
        plt.show()

        # Plot a histogram of the balanced cover types
        fig = plt.figure(figsize=(5, 3))
        plt.hist(y_res, color='orange', edgecolor='black')
        plt.ylabel('N')
        plt.xlabel('Cover Type')
        plt.title('Balanced Cover Types')
        plt.show()


    # ANN:
    print('Running the optimized ANN')
    # Create the ANN model with fine-tuned parameters
    clf_ANN = MLPClassifier(solver='adam', activation='relu', hidden_layer_sizes=(20,50,
                            max_iter=500, early_stopping=True, n_iter_no_change=5,
                            validation_fraction=0.1, learning_rate='adaptive', random_st

    # Fitting the model:
    clf_ANN.fit(X_tr_Norm, Y_tr)

    # predict the response for tr and va sets. We can have two outputs: probability (e.g
    ANN_tr_prob = clf_ANN.predict_proba(X_tr_Norm)
    ANN_tr_pred = clf_ANN.predict(X_tr_Norm)
    ANN_va_prob = clf_ANN.predict_proba(X_va_Norm)
    ANN_va_pred = clf_ANN.predict(X_va_Norm)

    # Create a class report and confusion matrix
    class_report_ann.append(classification_report(Y_va.argmax(axis=1), ANN_va_pred.argma
    cnf_matrix_ANN_tr.append(confusion_matrix(ANN_tr_pred.argmax(axis=1), Y_tr.argmax(ax
    cnf_matrix_ANN_va.append(confusion_matrix(ANN_va_pred.argmax(axis=1), Y_va.argmax(ax

    # SVM:
    print('Running the optimized SVM')
    # Initialize the fine-tuned model
    clf_svm = SVC(decision_function_shape='ovr', kernel='rbf', gamma=0.1, C=5, tol=1, pr

    # Fit the model on the data
```

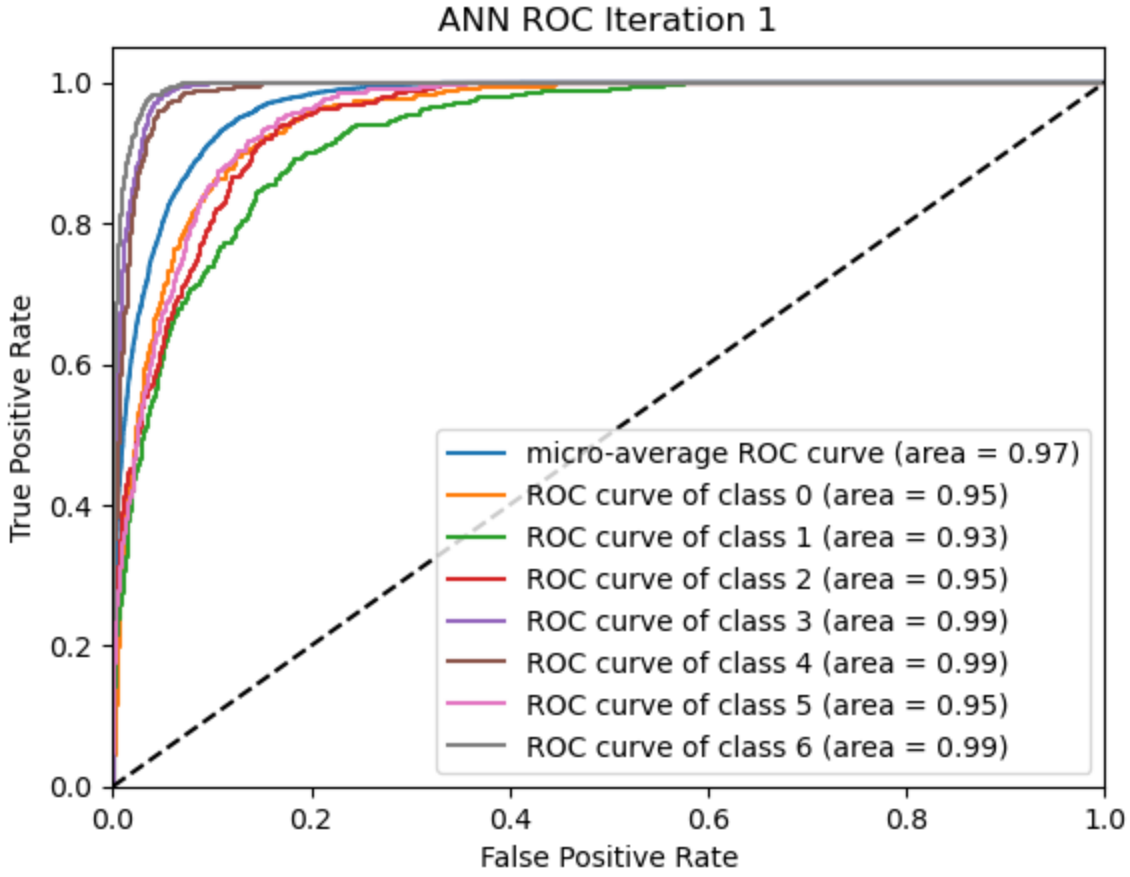
```
    clf_svm.fit(X_tr_Norm, Y_tr.argmax(axis=1))

    # Using those parameters, determine the training and validation predictions
    SVM_tr_prob = clf_svm.predict_proba(X_tr_Norm)
    SVM_tr_pred = clf_svm.predict(X_tr_Norm)
    SVM_va_prob = clf_svm.predict_proba(X_va_Norm)
    SVM_va_pred = clf_svm.predict(X_va_Norm)

    # Create a class report and confusion matrix
    class_report_svm.append(classification_report(Y_va.argmax(axis=1), SVM_va_pred, outp
    cnf_matrix_SVM_tr.append(confusion_matrix(SVM_tr_pred, Y_tr.argmax(axis=1)))
    cnf_matrix_SVM_va.append(confusion_matrix(SVM_va_pred, Y_va.argmax(axis=1)))

    # Plot the ROC curve for each iteration
    print('Plotting ROC Curves for Iteration ' + str(ii+1))
    plot_ROC_curve(ANN_va_prob, Y_va, title='ANN ROC Iteration ' + str(ii+1))
    plot_ROC_curve(SVM_va_prob, Y_va, title='SVM ROC Iteration ' + str(ii+1))

    print('Done iteration ' + str(ii+1) + ', woohoo!')
```

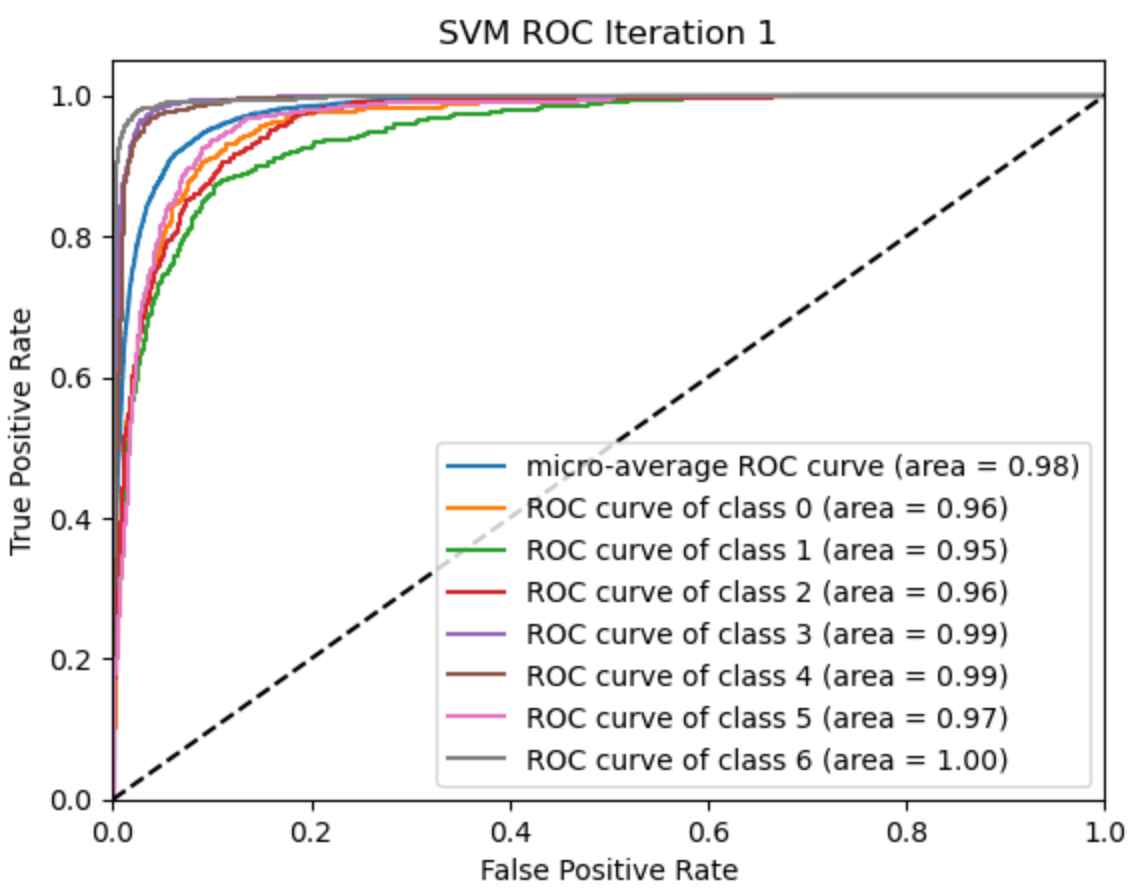
```
Starting iteration 1
Undersampling the data
Splitting the data
Normalizing the data
Plot of Normalized vs Not Normalized Data
```



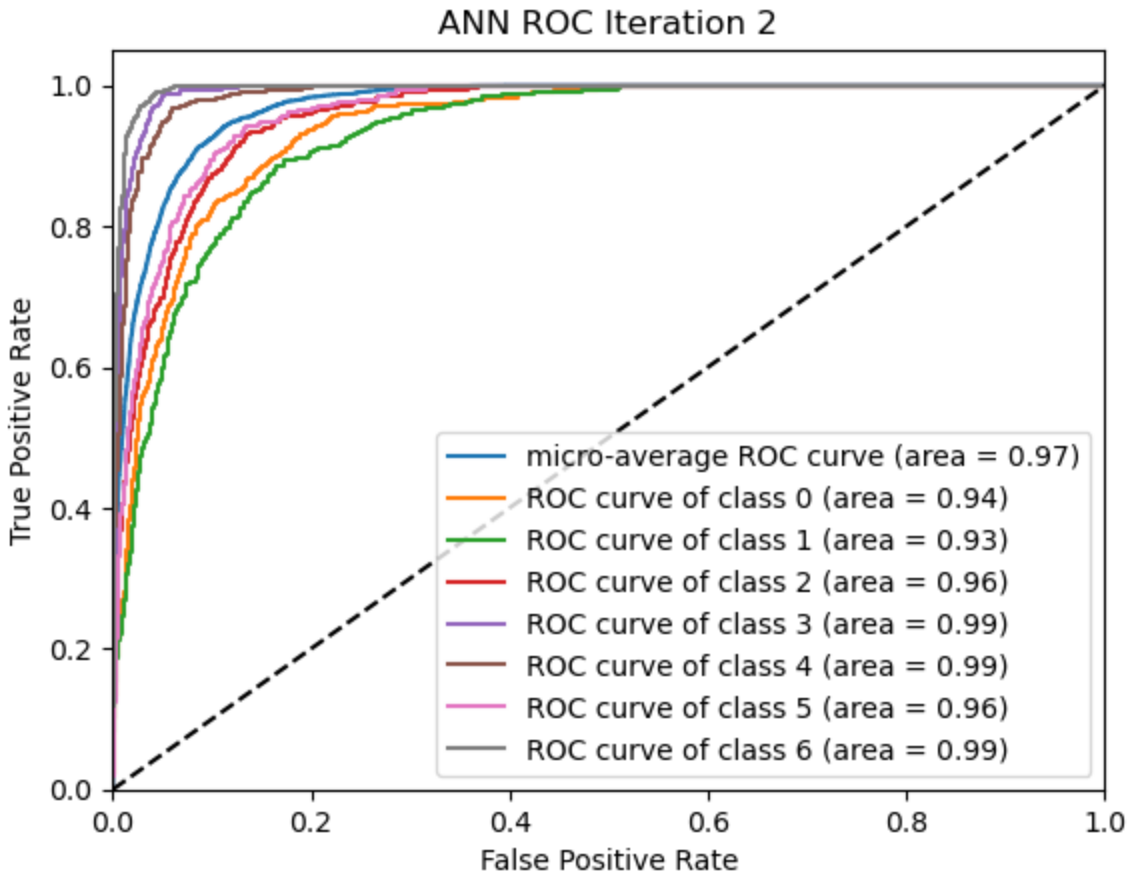


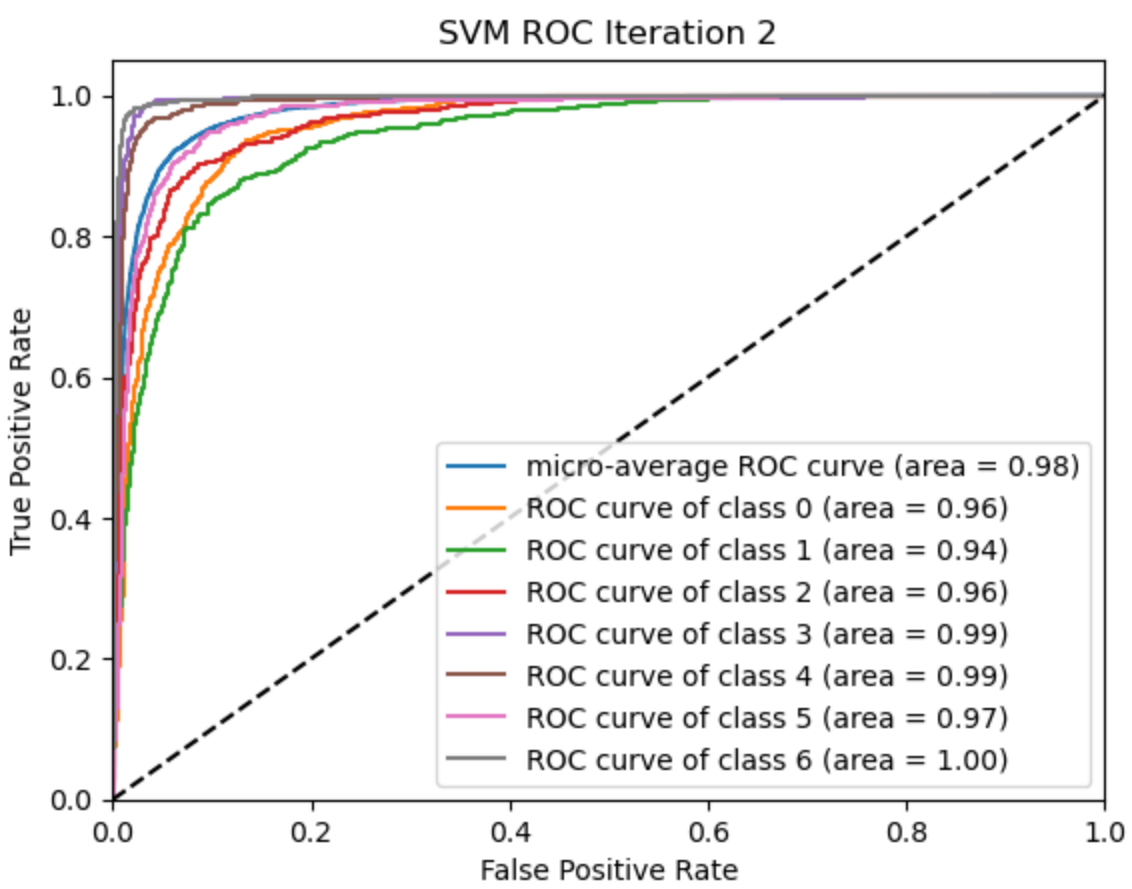

```
Running the optimized ANN
Running the optimized SVM
Plotting ROC Curves for Iteration 1
```



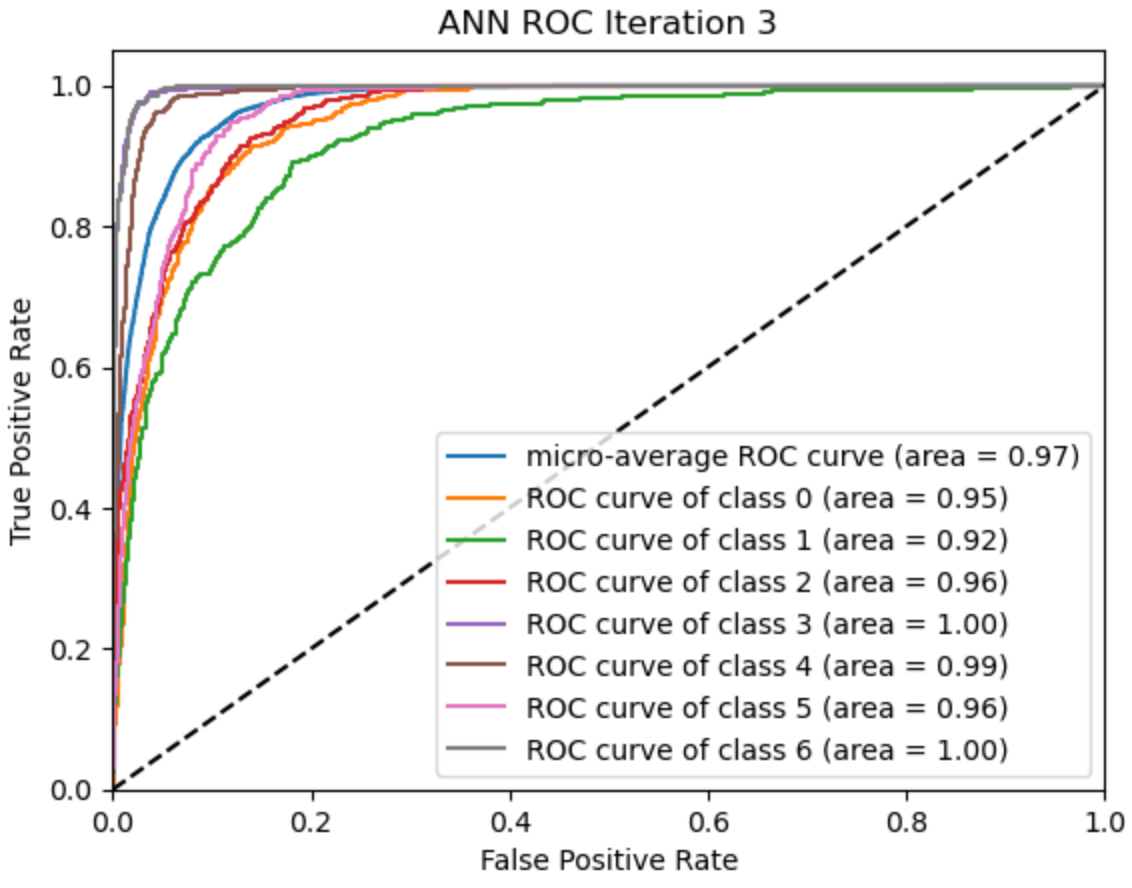


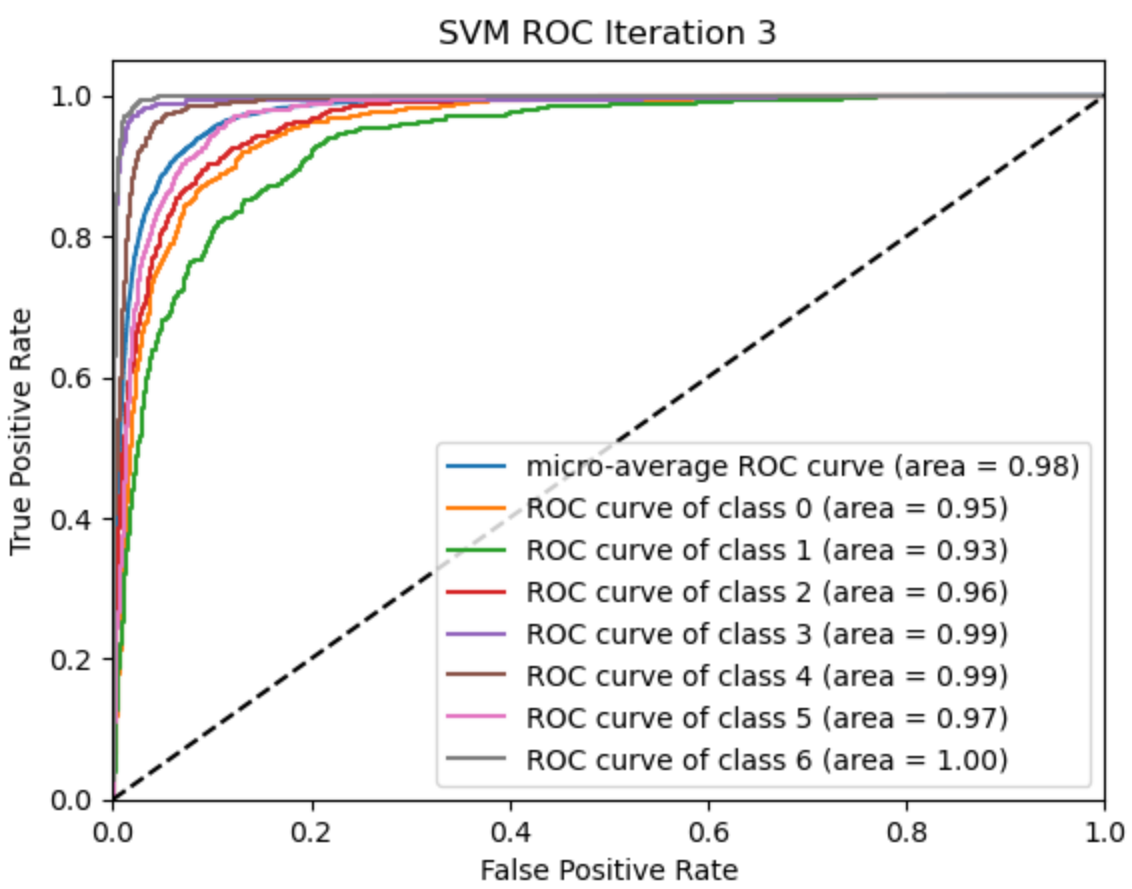

```
Done iteration 1, woohoo!
Starting iteration 2
Undersampling the data
Splitting the data
Normalizing the data
Running the optimized ANN
Running the optimized SVM
Plotting ROC Curves for Iteration 2
```



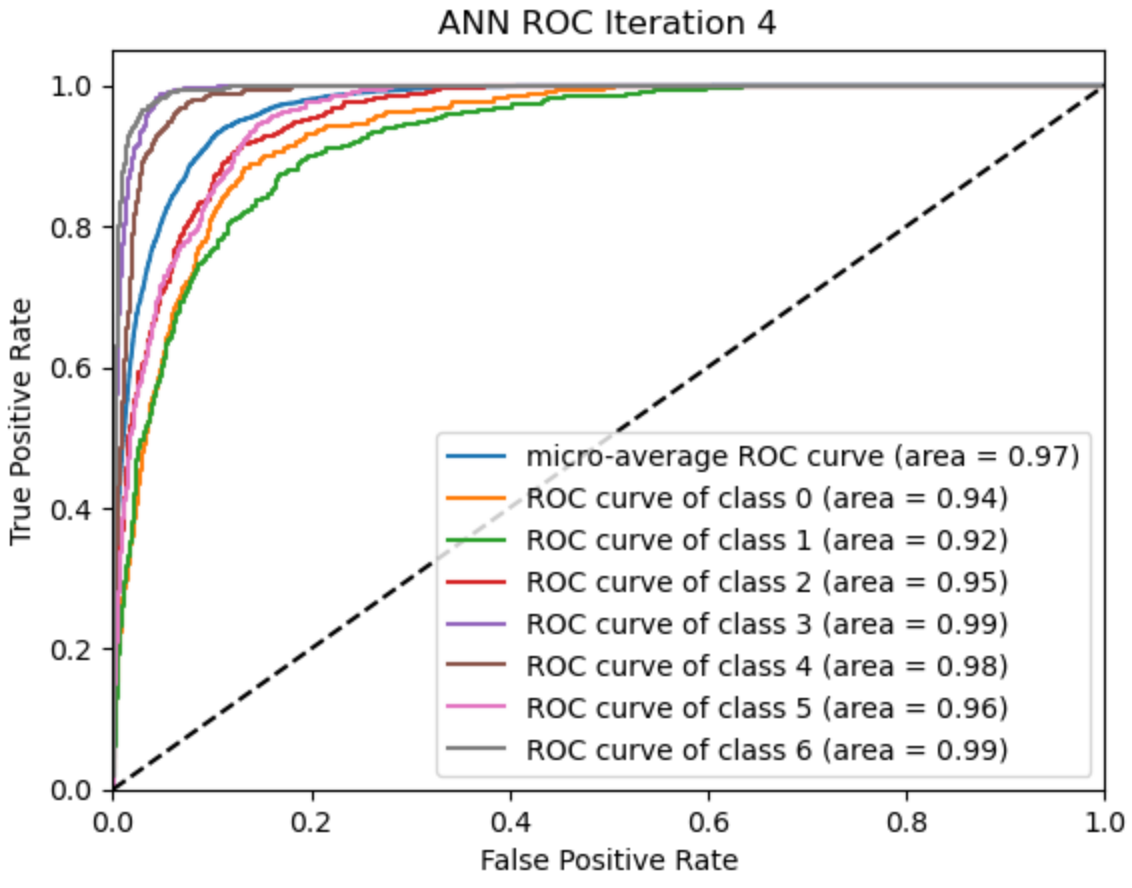


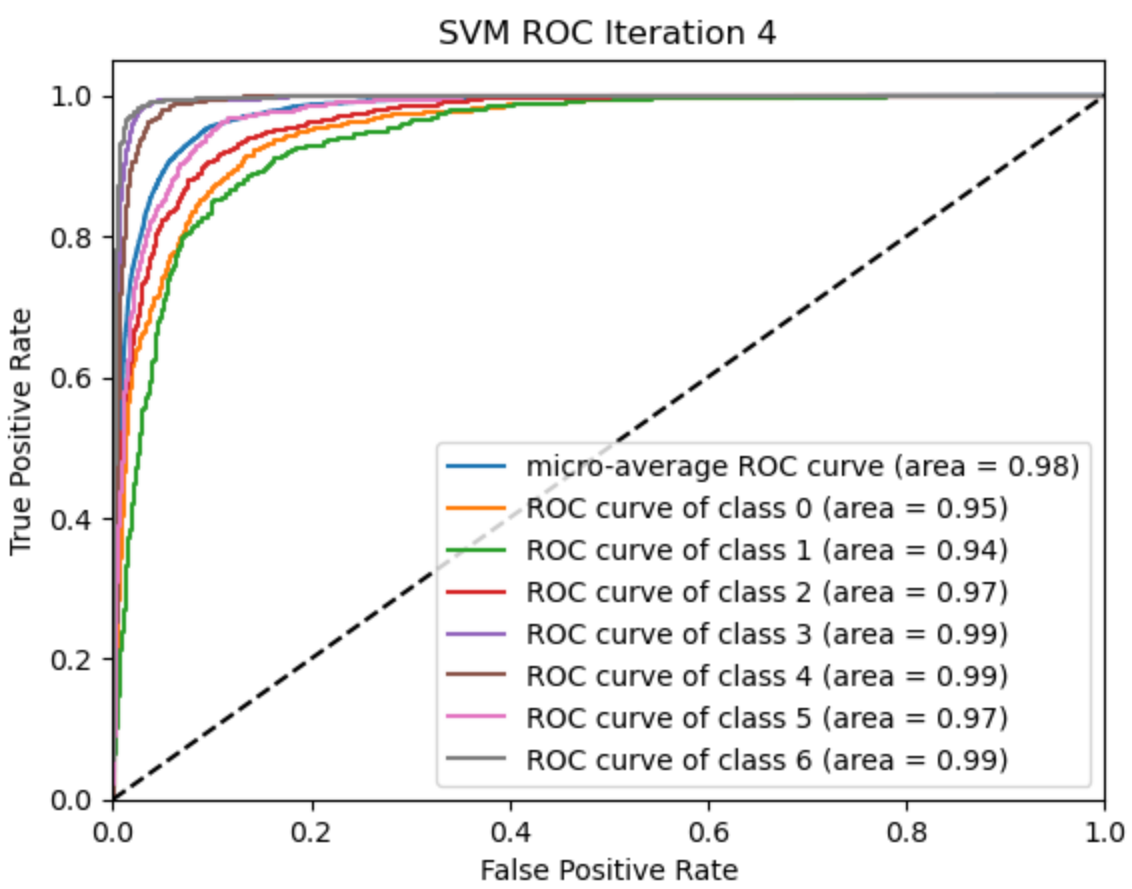

```
Done iteration 2, woohoo!
Starting iteration 3
Undersampling the data
Splitting the data
Normalizing the data
Running the optimized ANN
Running the optimized SVM
Plotting ROC Curves for Iteration 3
```



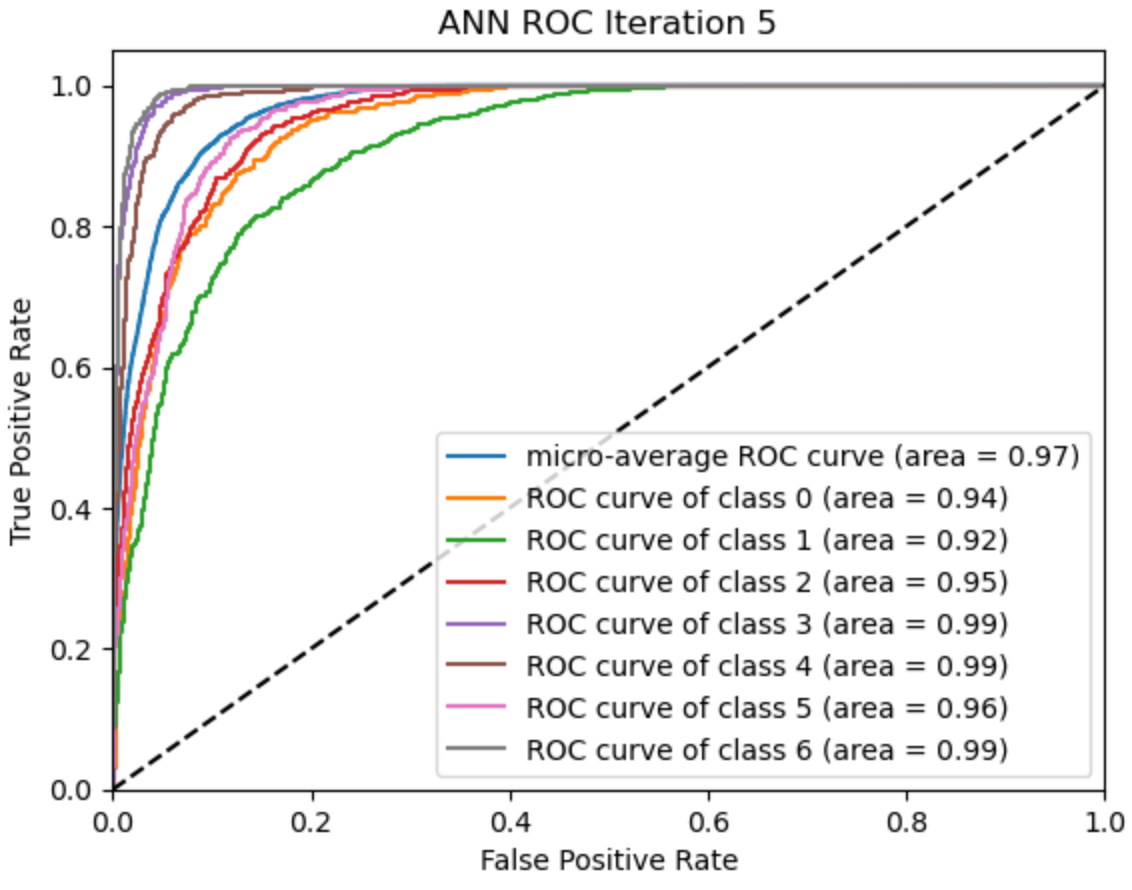


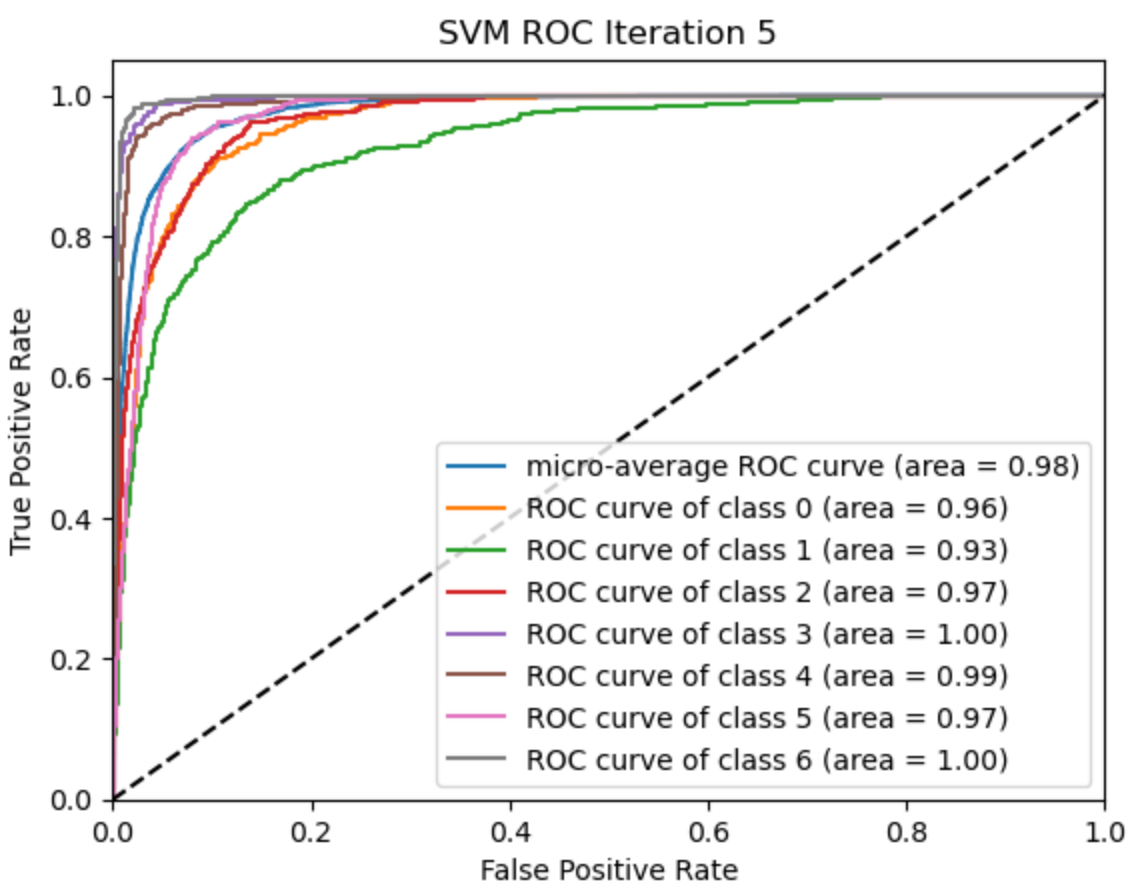

```
Done iteration 3, woohoo!
Starting iteration 4
Undersampling the data
Splitting the data
Normalizing the data
Running the optimized ANN
Running the optimized SVM
Plotting ROC Curves for Iteration 4
```






```
Done iteration 4, woohoo!
Starting iteration 5
Undersampling the data
Splitting the data
Normalizing the data
Running the optimized ANN
Running the optimized SVM
Plotting ROC Curves for Iteration 5
```






```
Done iteration 5, woohoo!
```

Background Calculations

In this section, I calculate the overall average result and the associated error. I have hidden the code as it is quite long.

In [10]:
```
# Calculate ANN results with error
class0_p_ann = []
class1_p_ann = []
class2_p_ann = []
class3_p_ann = []
class4_p_ann = []
class5_p_ann = []
class6_p_ann = []

class0_r_ann = []
class1_r_ann = []
class2_r_ann = []
class3_r_ann = []
class4_r_ann = []
class5_r_ann = []
class6_r_ann = []

class0_f_ann = []
class1_f_ann = []
class2_f_ann = []
class3_f_ann = []
class4_f_ann = []
class5_f_ann = []
class6_f_ann = []

accuracy_all_ann = []

n_class = 7
```

```
for ii in range(0, n_iter):
    for ii_2 in range(0, n_class):
        if ii_2 == 0:
            class0_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class0_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class0_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 1:
            class1_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class1_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class1_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 2:
            class2_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class2_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class2_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 3:
            class3_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class3_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class3_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 4:
            class4_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class4_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class4_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 5:
            class5_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class5_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class5_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 6:
            class6_p_ann.append(class_report_ann[ii-1][str(ii_2)]['precision'])
            class6_r_ann.append(class_report_ann[ii-1][str(ii_2)]['recall'])
            class6_f_ann.append(class_report_ann[ii-1][str(ii_2)]['f1-score'])

    accuracy_all_ann.append(class_report_ann[ii]['accuracy'])

# Calculate svm results with error

class0_p_svm = []
class1_p_svm = []
class2_p_svm = []
class3_p_svm = []
class4_p_svm = []
class5_p_svm = []
class6_p_svm = []

class0_r_svm = []
class1_r_svm = []
class2_r_svm = []
class3_r_svm = []
class4_r_svm = []
class5_r_svm = []
class6_r_svm = []

class0_f_svm = []
class1_f_svm = []
class2_f_svm = []
class3_f_svm = []
class4_f_svm = []
class5_f_svm = []
class6_f_svm = []

accuracy_all_svm = []

n_class = 7
for ii in range(0, n_iter):
    for ii_2 in range(0, n_class):
        if ii_2 == 0:
            class0_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
```

```
            class0_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class0_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 1:
            class1_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
            class1_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class1_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 2:
            class2_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
            class2_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class2_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 3:
            class3_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
            class3_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class3_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 4:
            class4_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
            class4_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class4_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 5:
            class5_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
            class5_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class5_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])
        if ii_2 == 6:
            class6_p_svm.append(class_report_svm[ii-1][str(ii_2)]['precision'])
            class6_r_svm.append(class_report_svm[ii-1][str(ii_2)]['recall'])
            class6_f_svm.append(class_report_svm[ii-1][str(ii_2)]['f1-score'])

    accuracy_all_svm.append(class_report_svm[ii]['accuracy'])

# Calculate the average and standard deviation for each class and metric
class0_p_svm_avg = np.mean(class0_p_svm)
class0_p_svm_std = np.std(class0_p_svm)
class1_p_svm_avg = np.mean(class1_p_svm)
class1_p_svm_std = np.std(class1_p_svm)
class2_p_svm_avg = np.mean(class2_p_svm)
class2_p_svm_std = np.std(class2_p_svm)
class3_p_svm_avg = np.mean(class3_p_svm)
class3_p_svm_std = np.std(class3_p_svm)
class4_p_svm_avg = np.mean(class4_p_svm)
class4_p_svm_std = np.std(class4_p_svm)
class5_p_svm_avg = np.mean(class5_p_svm)
class5_p_svm_std = np.std(class5_p_svm)
class6_p_svm_avg = np.mean(class6_p_svm)
class6_p_svm_std = np.std(class6_p_svm)
class0_p_ann_avg = np.mean(class0_p_ann)
class0_p_ann_std = np.std(class0_p_ann)
class1_p_ann_avg = np.mean(class1_p_ann)
class1_p_ann_std = np.std(class1_p_ann)
class2_p_ann_avg = np.mean(class2_p_ann)
class2_p_ann_std = np.std(class2_p_ann)
class3_p_ann_avg = np.mean(class3_p_ann)
class3_p_ann_std = np.std(class3_p_ann)
class4_p_ann_avg = np.mean(class4_p_ann)
class4_p_ann_std = np.std(class4_p_ann)
class5_p_ann_avg = np.mean(class5_p_ann)
class5_p_ann_std = np.std(class5_p_ann)
class6_p_ann_avg = np.mean(class6_p_ann)
class6_p_ann_std = np.std(class6_p_ann)
class0_p_svm_avg = np.mean(class0_p_svm)
class0_p_svm_std = np.std(class0_p_svm)
class1_p_svm_avg = np.mean(class1_p_svm)
class1_p_svm_std = np.std(class1_p_svm)
class2_p_svm_avg = np.mean(class2_p_svm)
class2_p_svm_std = np.std(class2_p_svm)
class3_p_svm_avg = np.mean(class3_p_svm)
class3_p_svm_std = np.std(class3_p_svm)
```

```
class4_p_svm_avg = np.mean(class4_p_svm)
class4_p_svm_std = np.std(class4_p_svm)
class5_p_svm_avg = np.mean(class5_p_svm)
class5_p_svm_std = np.std(class5_p_svm)
class6_p_svm_avg = np.mean(class6_p_svm)
class6_p_svm_std = np.std(class6_p_svm)
class0_r_ann_avg = np.mean(class0_r_ann)
class0_r_ann_std = np.std(class0_r_ann)
class1_r_ann_avg = np.mean(class1_r_ann)
class1_r_ann_std = np.std(class1_r_ann)
class2_r_ann_avg = np.mean(class2_r_ann)
class2_r_ann_std = np.std(class2_r_ann)
class3_r_ann_avg = np.mean(class3_r_ann)
class3_r_ann_std = np.std(class3_r_ann)
class4_r_ann_avg = np.mean(class4_r_ann)
class4_r_ann_std = np.std(class4_r_ann)
class5_r_ann_avg = np.mean(class5_r_ann)
class5_r_ann_std = np.std(class5_r_ann)
class6_r_ann_avg = np.mean(class6_r_ann)
class6_r_ann_std = np.std(class6_r_ann)
class0_r_svm_avg = np.mean(class0_r_svm)
class0_r_svm_std = np.std(class0_r_svm)
class1_r_svm_avg = np.mean(class1_r_svm)
class1_r_svm_std = np.std(class1_r_svm)
class2_r_svm_avg = np.mean(class2_r_svm)
class2_r_svm_std = np.std(class2_r_svm)
class3_r_svm_avg = np.mean(class3_r_svm)
class3_r_svm_std = np.std(class3_r_svm)
class4_r_svm_avg = np.mean(class4_r_svm)
class4_r_svm_std = np.std(class4_r_svm)
class5_r_svm_avg = np.mean(class5_r_svm)
class5_r_svm_std = np.std(class5_r_svm)
class6_r_svm_avg = np.mean(class6_r_svm)
class6_r_svm_std = np.std(class6_r_svm)
class0_f_ann_avg = np.mean(class0_f_ann)
class0_f_ann_std = np.std(class0_f_ann)
class1_f_ann_avg = np.mean(class1_f_ann)
class1_f_ann_std = np.std(class1_f_ann)
class2_f_ann_avg = np.mean(class2_f_ann)
class2_f_ann_std = np.std(class2_f_ann)
class3_f_ann_avg = np.mean(class3_f_ann)
class3_f_ann_std = np.std(class3_f_ann)
class4_f_ann_avg = np.mean(class4_f_ann)
class4_f_ann_std = np.std(class4_f_ann)
class5_f_ann_avg = np.mean(class5_f_ann)
class5_f_ann_std = np.std(class5_f_ann)
class6_f_ann_avg = np.mean(class6_f_ann)
class6_f_ann_std = np.std(class6_f_ann)
class0_f_svm_avg = np.mean(class0_f_svm)
class0_f_svm_std = np.std(class0_f_svm)
class1_f_svm_avg = np.mean(class1_f_svm)
class1_f_svm_std = np.std(class1_f_svm)
class2_f_svm_avg = np.mean(class2_f_svm)
class2_f_svm_std = np.std(class2_f_svm)
class3_f_svm_avg = np.mean(class3_f_svm)
class3_f_svm_std = np.std(class3_f_svm)
class4_f_svm_avg = np.mean(class4_f_svm)
class4_f_svm_std = np.std(class4_f_svm)
class5_f_svm_avg = np.mean(class5_f_svm)
class5_f_svm_std = np.std(class5_f_svm)
class6_f_svm_avg = np.mean(class6_f_svm)
class6_f_svm_std = np.std(class6_f_svm)
accuracy_all_ann_avg = np.mean(accuracy_all_ann)
accuracy_all_ann_std = np.std(accuracy_all_ann)
accuracy_all_svm_avg = np.mean(accuracy_all_svm)
accuracy_all_svm_std = np.std(accuracy_all_svm)
```

```
#create tabular results
data = [["1", class0_p_ann_avg, class0_p_ann_std, class0_r_ann_avg, class0_r_ann_std, cl
        ["2", class1_p_ann_avg, class1_p_ann_std, class1_r_ann_avg, class1_r_ann_std, cl
       ["3", class2_p_ann_avg, class2_p_ann_std, class2_r_ann_avg, class2_r_ann_std, cla
       ["4", class3_p_ann_avg, class3_p_ann_std, class3_r_ann_avg, class3_r_ann_std, cla
       ["5", class4_p_ann_avg, class4_p_ann_std, class4_r_ann_avg, class4_r_ann_std, cla
       ["6", class5_p_ann_avg, class5_p_ann_std, class5_r_ann_avg, class5_r_ann_std, cla
       ["7", class6_p_ann_avg, class6_p_ann_std, class6_r_ann_avg, class6_r_ann_std, cla

#define header names
col_names = ["Class", "Precision", "Std Precision", "Recall", "Std Recall", "F1", "Std F

#create data
data_svm = [["1", class0_p_svm_avg, class0_p_svm_std, class0_r_svm_avg, class0_r_svm_std
        ["2", class1_p_svm_avg, class1_p_svm_std, class1_r_svm_avg, class1_r_svm_std, cl
       ["3", class2_p_svm_avg, class2_p_svm_std, class2_r_svm_avg, class2_r_svm_std, cla
       ["4", class3_p_svm_avg, class3_p_svm_std, class3_r_svm_avg, class3_r_svm_std, cla
       ["5", class4_p_svm_avg, class4_p_svm_std, class4_r_svm_avg, class4_r_svm_std, cla
       ["6", class5_p_svm_avg, class5_p_svm_std, class5_r_svm_avg, class5_r_svm_std, cla
       ["7", class6_p_svm_avg, class6_p_svm_std, class6_r_svm_avg, class6_r_svm_std, cla

#define header names
col_names_svm = ["Class", "Precision", "Std Precision", "Recall", "Std Recall", "F1", "S

# add up all of the confusion matrices (can do this because I will use the normalized co
all_conf_ANN_tr = cnf_matrix_ANN_tr[0]
for conf in range(1,n_iter):
    all_conf_ANN_tr += cnf_matrix_ANN_tr[conf]

all_conf_ANN_va = cnf_matrix_ANN_va[0]
for conf in range(1,n_iter):
    all_conf_ANN_va += cnf_matrix_ANN_va[conf]

all_conf_SVM_tr = cnf_matrix_SVM_tr[0]
for conf in range(1,n_iter):
    all_conf_SVM_tr += cnf_matrix_SVM_tr[conf]

all_conf_SVM_va = cnf_matrix_SVM_va[0]
for conf in range(1,n_iter):
    all_conf_SVM_va += cnf_matrix_SVM_va[conf]
```

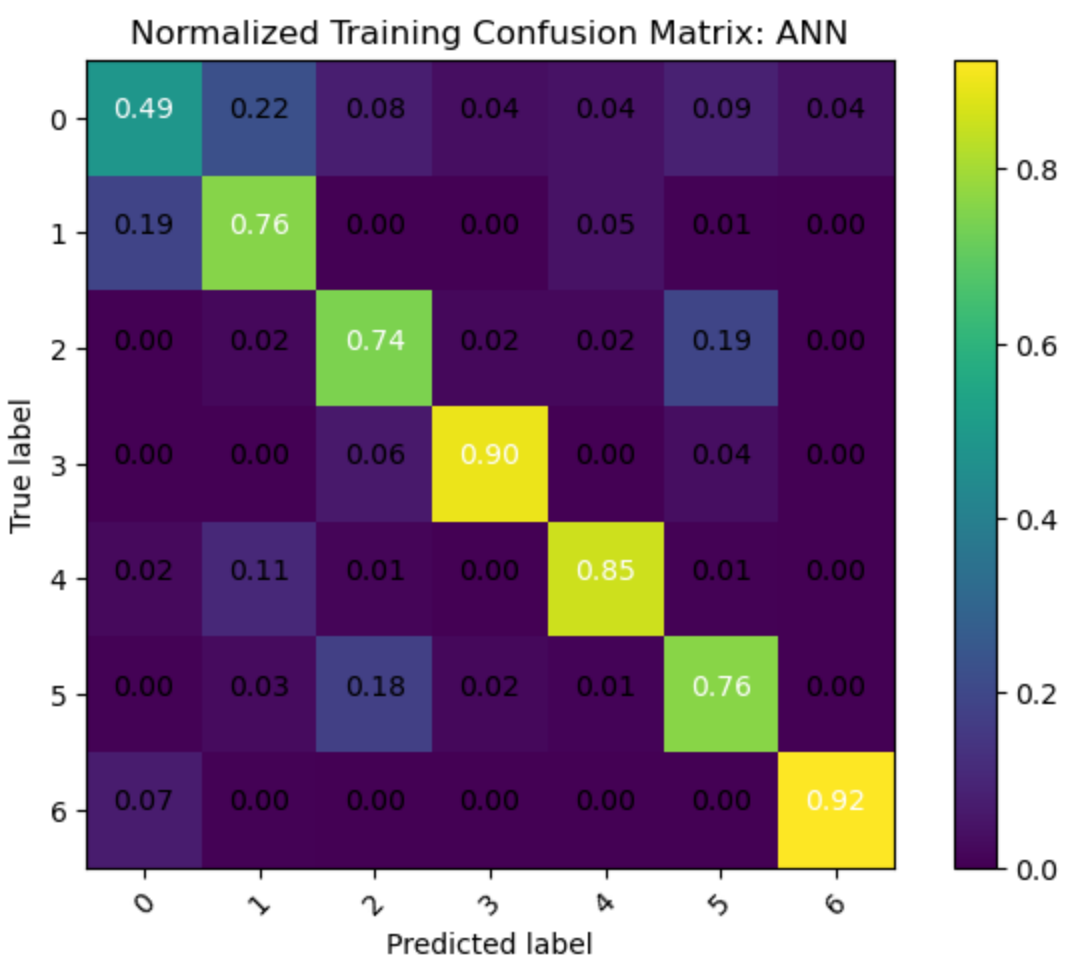
Results

I present the average confusion matrix for the training and validation sets (ie. all iterations added together, and normalized), for both the ANN and SVM algorithms. I also present tabular results for precision, accuracy, F1 score, and recall for both algorithms with associated error (standard deviation).

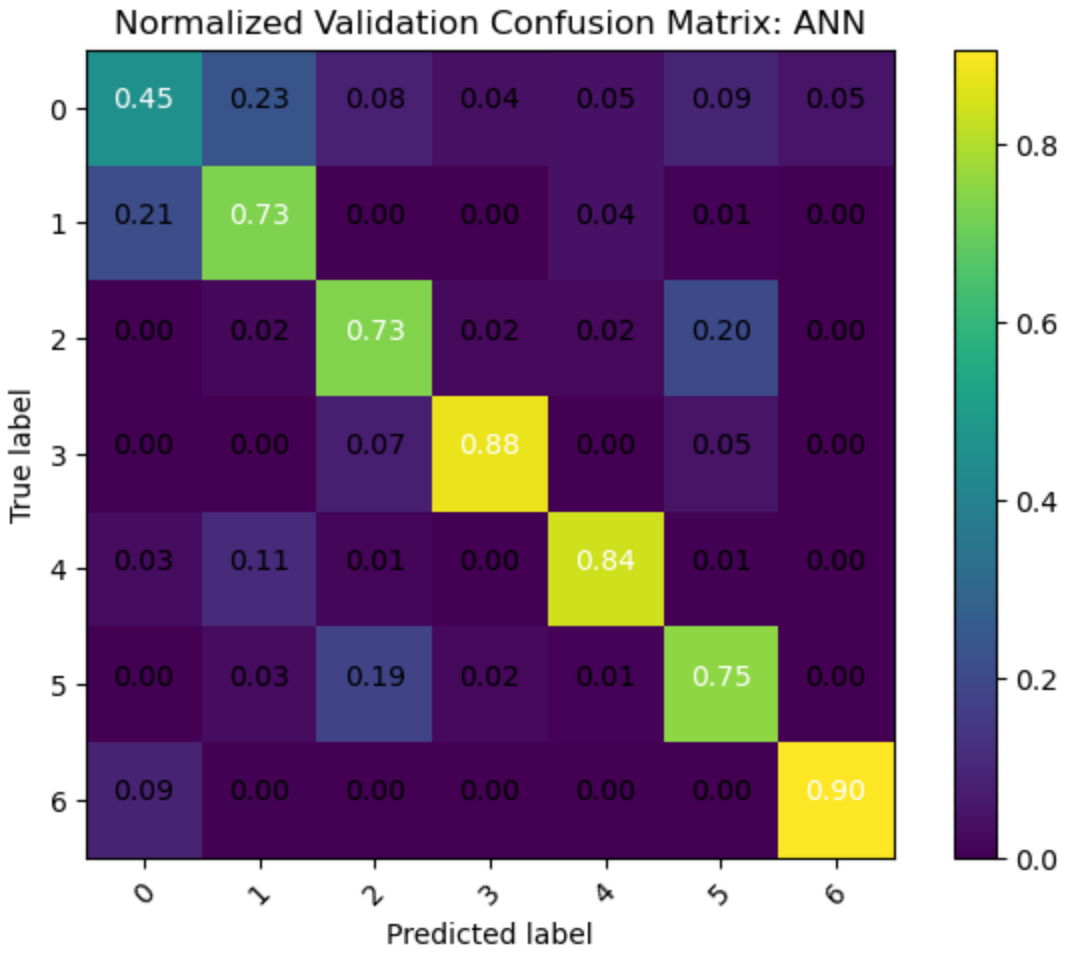
Confusion Matrices

In [11]:
```
plot_confusion_matrix(all_conf_ANN_tr, classes=['0','1', '2', '3', '4', '5', '6'], norma
                      title='Normalized Training Confusion Matrix: ANN')

plot_confusion_matrix(all_conf_ANN_va, classes=['0','1', '2', '3', '4', '5', '6'], norma
                      title='Normalized Validation Confusion Matrix: ANN')
```

```
Normalized confusion matrix
```



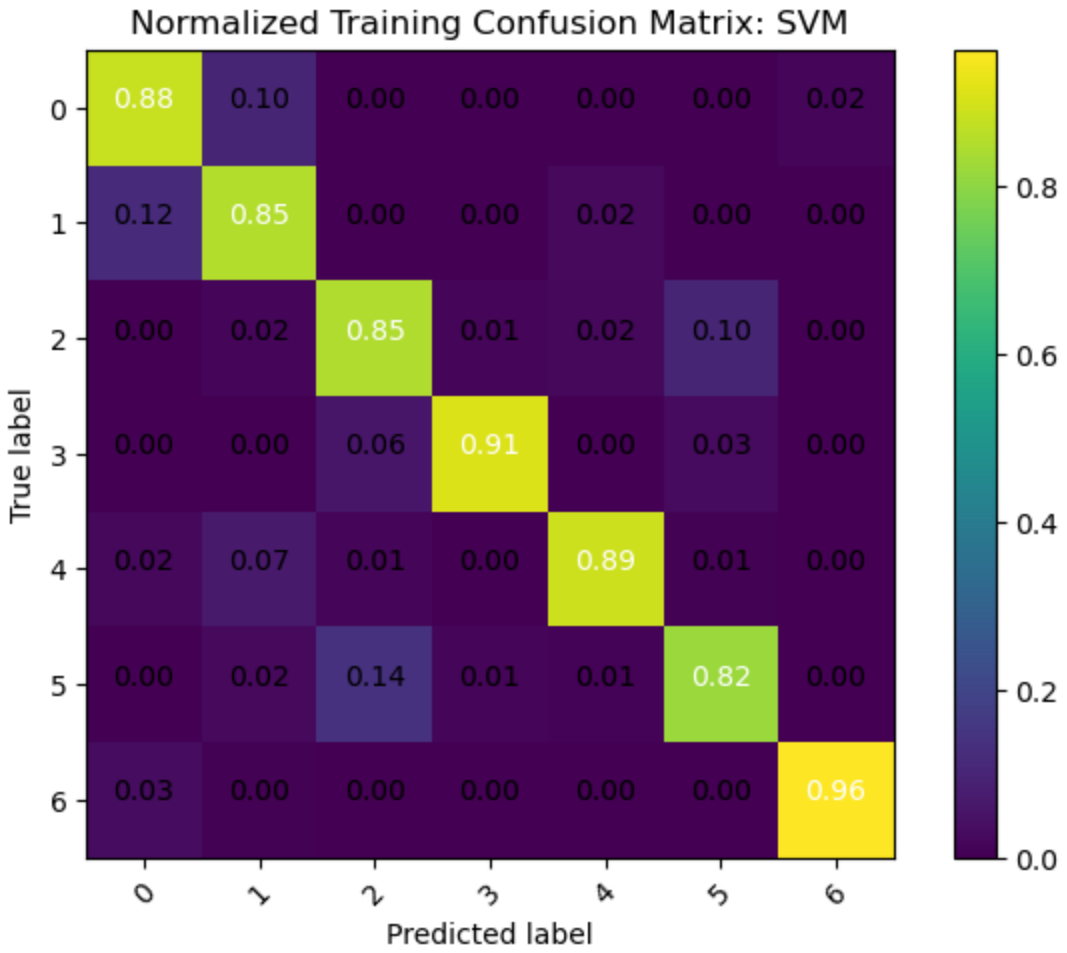
Normalized confusion matrix



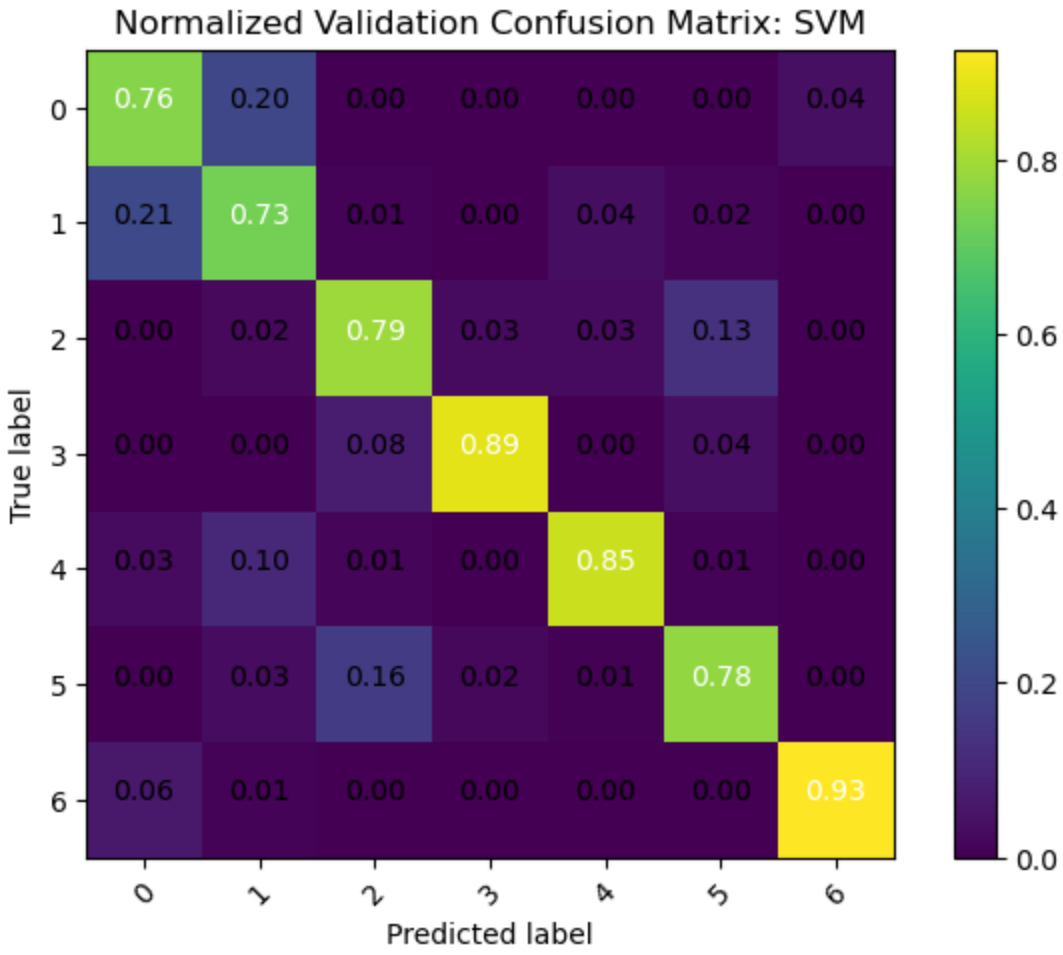
In [12]:
```
plot_confusion_matrix(all_conf_SVM_tr, classes=['0','1', '2', '3', '4', '5', '6'], norma
                      title='Normalized Training Confusion Matrix: SVM')
```

```
plot_confusion_matrix(all_conf_SVM_va, classes=['0','1', '2', '3', '4', '5', '6'], norma
                      title='Normalized Validation Confusion Matrix: SVM')
```

```
Normalized confusion matrix
```




```
Normalized confusion matrix
```



Precision, Recall, F1, Accuracy, and Standard Deviations

In [13]:
```
print(str(n_iter) + ' Runs of Randomly Undersampled Shuffled Data')
print('')
#display table
print('ANN Results')
print(tabulate(data, headers=col_names, tablefmt="fancy_grid"))
print('Accuracy of ANN is ' + str(round(accuracy_all_ann_avg, 2)) + ' with standard devi
print('')
#display table
print('SVM Results')
print(tabulate(data_svm, headers=col_names_svm, tablefmt="fancy_grid"))
print('Accuracy of SVM is ' + str(round(accuracy_all_svm_avg, 2)) + ' with standard devi
```

```
5 Runs of Randomly Undersampled Shuffled Data

ANN Results
```

Class	Precision	Std Precision	Recall	Std Recall	F1	Std F1
1	0.454204	0.0247553	0.7407	0.0167739	0.562894	0.0232624
2	0.732099	0.021462	0.461011	0.0133428	0.565499	0.0109829
3	0.733778	0.0190772	0.632689	0.0650881	0.676772	0.0339165

Class	Precision	Std Precision	Recall	Std Recall	F1	Std F1
4	0.88005	0.0302571	0.891248	0.0153831	0.885189	0.0144802
5	0.839351	0.010765	0.872727	0.0239835	0.855492	0.0119729
6	0.75752	0.0426127	0.623944	0.0407539	0.681825	0.0109829
7	0.904949	0.00767256	0.913143	0.00856931	0.908984	0.00513144

Accuracy of ANN is 0.73 with standard deviation of 0.01

SVM Results

Class	Precision	Std Precision	Recall	Std Recall	F1	Std F1
1	0.75803	0.0109673	0.719337	0.0200593	0.737988	0.0115422
2	0.73362	0.0162678	0.641155	0.01932	0.684189	0.0166069
3	0.791545	0.0146303	0.730404	0.0185194	0.759579	0.0127532
4	0.886892	0.0167368	0.951955	0.00581769	0.918181	0.0091808
5	0.850501	0.0109969	0.932364	0.00556257	0.889537	0.00829913
6	0.777893	0.0156628	0.818662	0.0233565	0.797519	0.0166069
7	0.926057	0.00407733	0.958857	0.00728805	0.942158	0.00462826

Accuracy of SVM is 0.82 with standard deviation of 0.004

Conclusion (Classification: ANN vs SVM)

(The numbers in the below discussion might change slightly if one was to rerun the notebook, due to the undersampler random seed not being set. This was intentional to ensure the data was sampled differently randomly each time.)

The accuracy of the ANN was 73% +- 0.01, and the accuracy of the SVM was 82% +- 0.004, showing an almost 10% increase in classification accuracy for the SVM. One noteable result was the precision for class 1 between the ANN and SVM. The ANN suffered greatly, having a precision of 45% +- 0.025, whereas the SVM had a precision of 76% +- 0.011. This is a 30% increase in precision! For this data set and these algorithms, the SVM consistently outperforms the ANN, as shown by the ROC curves for each iteration, the confusion matrices, and tabular results.

Note that it is important to recognize that these results might change if a more complicated neural network was used, instead of the "simple" 3 layer network used here. This structure was chosen due to constraints on time and computational resources, but was optimized using a pipeline, similar to how the SVM was optimized.

For these specific algorithms and this data set, the SVM classifier would be an appropriate choice for future classifications.

Regression

In this section, I apply an ANN and a SVM to the "Sachs Harbour Data set", from Insley, S. et. al. (2017). Seasonal Patterns in Ocean Ambient Noise near Sachs Harbour, Northwest Territories. Arctic, 70(3), 239–248. https://doi.org/10.14430/arctic4662.

This is an acoustical data set, with ambient sound pressure level (SPL), environmental variables, and ice concentration as attributes. As climate change continues to warm our planet, the Arctic sea ice continues to melt, increasing ambient sound levels and anthropogenic sound levels. This paper attempted to provide a baseline for ambient sound levels, so that we might be able to detect any future changes.

The target of regression in this analysis is Ice Concentration at 106km^2 from the hydrophone.

Functions

In [14]:
```
# Regression stats plots
def stats_plots(Y_tr, Y_va, Y_tr_pred, Y_va_pred, xlim, ylim, textx, texty, text2x, text

    #Statistical information regarding training and validation predictions
    mu = np.mean(Y_tr-Y_tr_pred)
    median = np.median(Y_tr-Y_tr_pred)
    sigma = np.std(Y_tr-Y_tr_pred)

    muv = np.mean(Y_va-Y_va_pred)
    medianv = np.median(Y_va-Y_va_pred)
    sigmav = np.std(Y_va-Y_va_pred)

    textstr = '$\mu=%.4f$\n$\mathrm{med}=%.4f$\n$\sigma=%.4f$'%(mu, median, sigma)
    textstrv = '$\mu=%.4f$\n$\mathrm{med}=%.4f$\n$\sigma=%.4f$'%(muv, medianv, sigmav)

    plt.figure(1)
    plt.plot(Y_tr,Y_tr_pred,'ob')
    plt.plot(Y_va,Y_va_pred,'.r')

    plt.plot(np.arange(xlim,ylim,.1),  np.arange(xlim,ylim,.1),'-k')
    plt.xlabel(str(x_var))
    plt.ylabel('Predicted ' + str(x_var))
    plt.legend(['Training', 'Validation'], loc='best')
```

```
    plt.text(textx,texty,textstr, color='b',fontsize=15)
    plt.text(text2x,text2y,textstrv, color='r',fontsize=15)
    plt.title(str(title))

    plt.figure(2)
    plt.hist(Y_tr_pred-Y_tr,20,color='b',histtype='step',density=True,label='training')
    plt.hist(Y_va_pred-Y_va,20,color='r',histtype='step',density=True,label='validation'
    plt.xlabel('Predicted - real')
    plt.xlim([-20, 20])
    plt.ylabel('Probability (density)')
    plt.legend(loc='upper left')

    plt.figure(3)
    plt.scatter(Y_tr,Y_tr_pred-Y_tr,label='training',color='b',alpha=.3)
    plt.scatter(Y_va,Y_va_pred-Y_va,label='test',color='r',alpha=.3)
    plt.xlabel('Y (real)')
    plt.ylim([-30,30])
    plt.ylabel('Y (predicted) - Y (real)')
    plt.plot([xlim,ylim],[0,0],'y')
    plt.legend(loc='best')
```

Load the data

In [15]:
```
# Load in the data
df_SH = pd.read_excel('Ambient Sound Data Sachs Harbour 2015-2016.xlsx')
print('The shape of the dataframe is ' + str(df_SH.shape))

# Count the existing nans and drop rows containing nans
nan_count = df_SH.isna().sum().sum()
print('The number of nans in the dataset is ' + str(nan_count))
df_SH=df_SH.dropna(axis=0)
print('The shape of the dataframe after dropping NaNs is ' + str(df_SH.shape))

# Print the first few rows of the data
df_SH.head()
```

```
The shape of the dataframe is (9982, 23)
The number of nans in the dataset is 564
The shape of the dataframe after dropping NaNs is (9642, 23)
```

Out[15]:

	Deployment	Year	Month	Day	Hour	DateTime	10-100Hz	100-1000Hz	1-10kHz	10-24kHz	...	DewPt	Rel
0	1	2015	5	18	15	2015-05-18 15:00:00	86.113543	83.540120	75.726148	65.178992	...	-0.1	
1	1	2015	5	18	16	2015-05-18 16:00:00	86.100652	96.371012	89.047766	70.179126	...	0.1	
2	1	2015	5	18	17	2015-05-18 17:00:00	69.981675	98.985918	91.915703	65.467056	...	0.6	
3	1	2015	5	18	18	2015-05-18 18:00:00	68.305432	97.577544	91.349444	65.249551	...	1.4	
4	1	2015	5	18	19	2015-05-18 19:00:00	68.952387	98.386537	91.631783	65.269117	...	1.1	

5 rows × 23 columns

Above is a table of the variables within the data set. I drop "deployment", "datetime", and "ice" variables to continue with the analysis as they are not appropriate for regression. Ambient sound levels are presented in specific frequency bands. Dew point, relative humidity, ..., and pressure are variables from Environment Canada. Ice concentrations are satellite data, given as percentages of cover at certain radii away from the hydrophone. I chose ice at 106km^2 as it is the largest. "Ice" is a binary column indicating if the ice concentration was greater than 50%, and this variable was added by a previous analyst when looking at classification of when ice would form. For regression, this column is unnecessary.

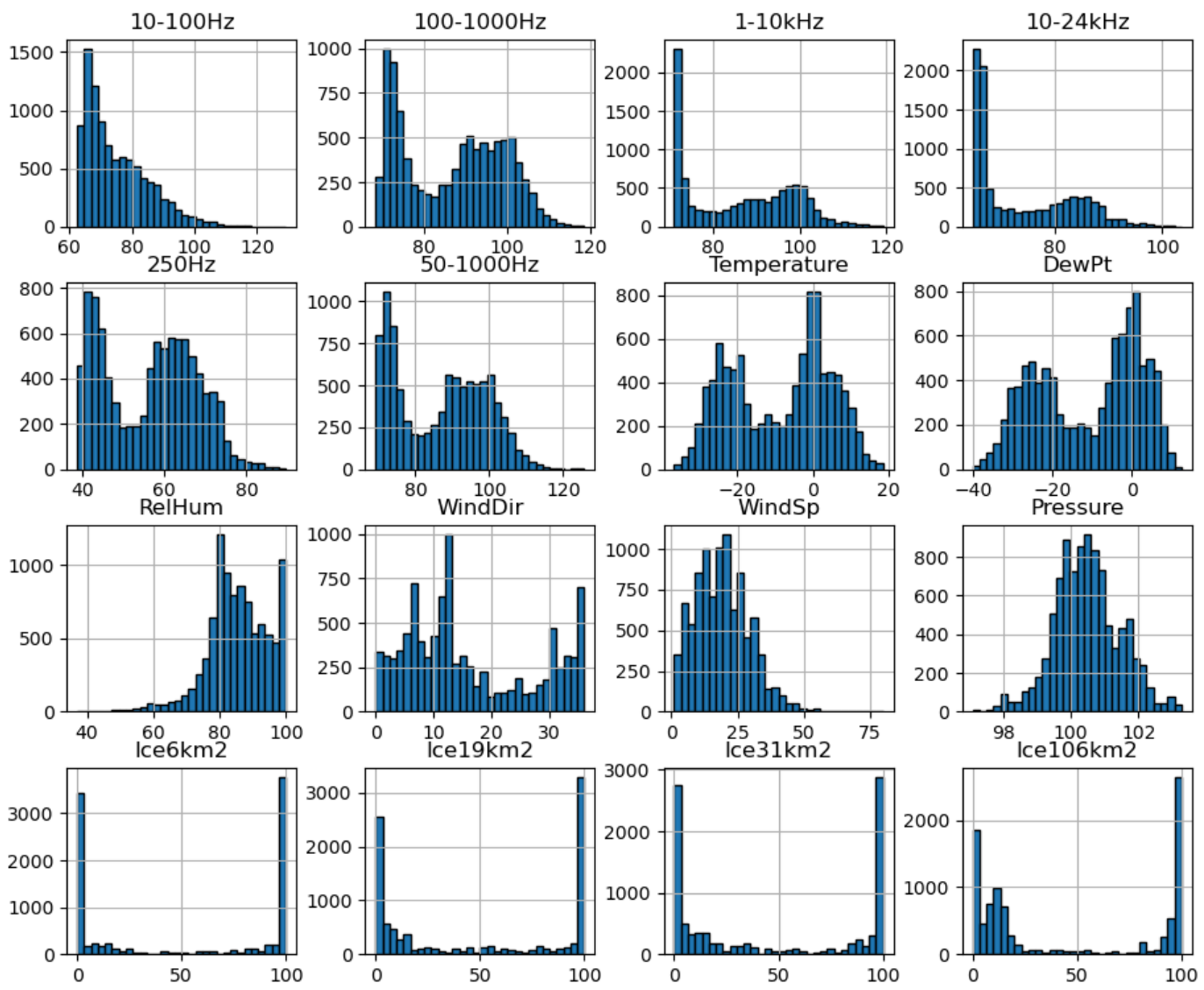
In [16]:
```
# Drop non necessary columns
df_SH = df_SH.drop(['Deployment','DateTime', 'Ice'], axis=1)

# Create dataframe for histogram plotting
df_hist = df_SH.drop(['Month', 'Day', 'Year', 'Hour'], axis=1)

# Set Ice Concentration as the target variable
var = ['Ice106km2']

# Drop temperature out of the weather data, and set it as X
X = df_SH.loc[:,df_SH.columns.drop(var)]
print(X.columns)

# Set the target (Y) to be temperature
Y = df_SH[var].copy()
print(Y.columns)

# Call the split data function
X_tr, X_va, Y_tr, Y_va = train_test_split(X,Y.values.ravel(),test_size=0.25)

# Print the shape of the split data
print ('training set == ',np.shape(X_tr),np.shape(Y_tr),',, validation set == ', np.shap

# Normalize the data, as was done in Q2
scaler_S= StandardScaler().fit(X_tr)  # line #2
X_tr_Norm= scaler_S.transform(X_tr) # line # 3
X_va_Norm= scaler_S.transform(X_va)  # Line #4
```

```
Index(['Year', 'Month', 'Day', 'Hour', '10-100Hz', '100-1000Hz', '1-10kHz',
       '10-24kHz', '250Hz', '50-1000Hz', 'Temperature', 'DewPt', 'RelHum',
       'WindDir', 'WindSp', 'Pressure', 'Ice6km2', 'Ice19km2', 'Ice31km2'],
      dtype='object')
Index(['Ice106km2'], dtype='object')
training set ==  (7231, 19) (7231,) ,, validation set ==  (2411, 19) (2411,)
```

Visualize the data

In [17]:
```
# Plot a histogram of the data
df_hist.hist(figsize=(11, 9), bins=30, edgecolor="black")
print(df_SH.shape)
```

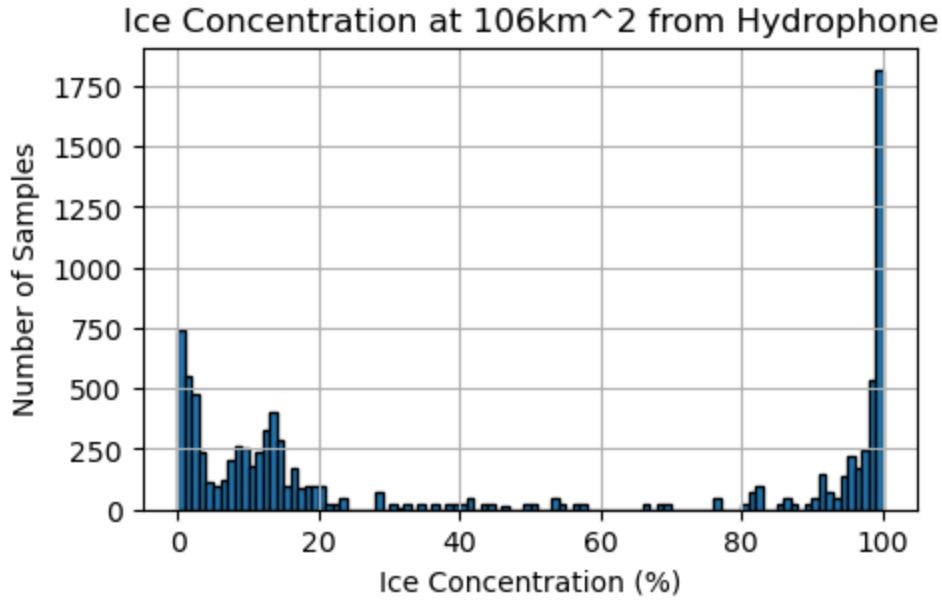
```
(9642, 20)
```



Above are histograms for all continuous variables.

In [18]:
```
df_hist.Ice106km2.hist(figsize=(5, 3), bins=100, edgecolor="black")
plt.title('Ice Concentration at 106km^2 from Hydrophone')
plt.xlabel('Ice Concentration (%)')
plt.ylabel('Number of Samples')
```

Out[18]:
```
Text(0, 0.5, 'Number of Samples')
```

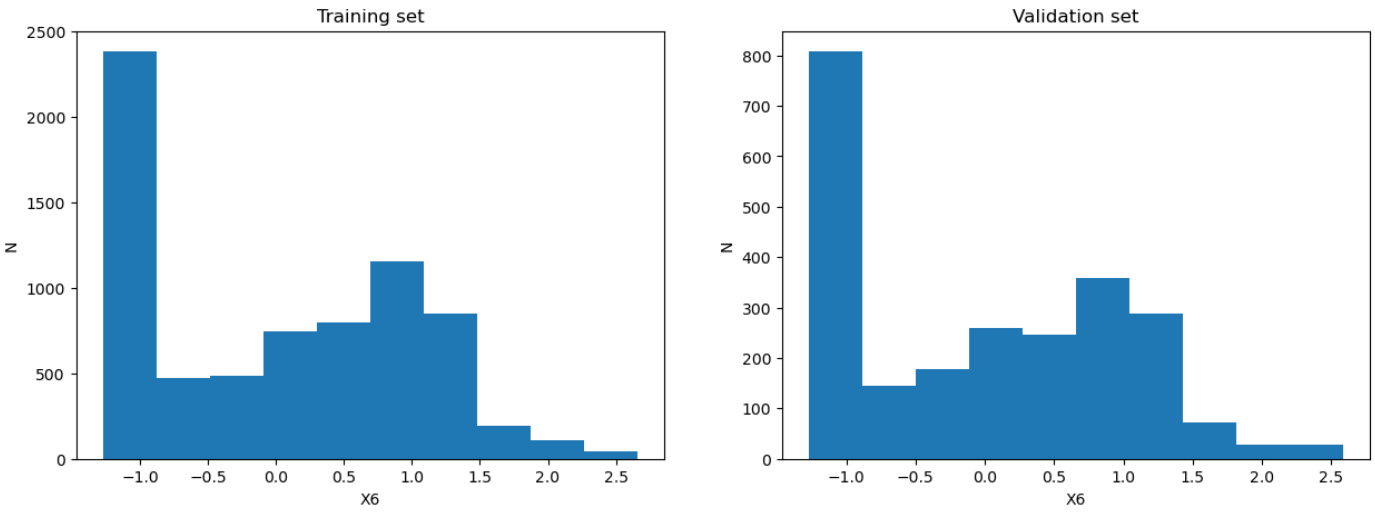


Looking closer at the target variable, there is more data near the 100% coverage and 0% coverage than other values. This is because this is a shore station, where ice forms and breaks up quickly, so most of the year is either covered in ice, or open water. This would look different in an open water situation, with a more spread out distribution.

Below I plot the normalized vs. unnormalized data, using the 10-100Hz frequency band as an example.

In [19]:
```
fig = plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plt.hist(X_tr_Norm[:,6])
plt.title('Training set')
plt.ylabel('N')
plt.xlabel("X"+str(6))

plt.subplot(1, 2, 2)
plt.hist(X_va_Norm[:,6])
plt.title('Validation set')
plt.ylabel('N')
plt.xlabel("X"+str(6))
```

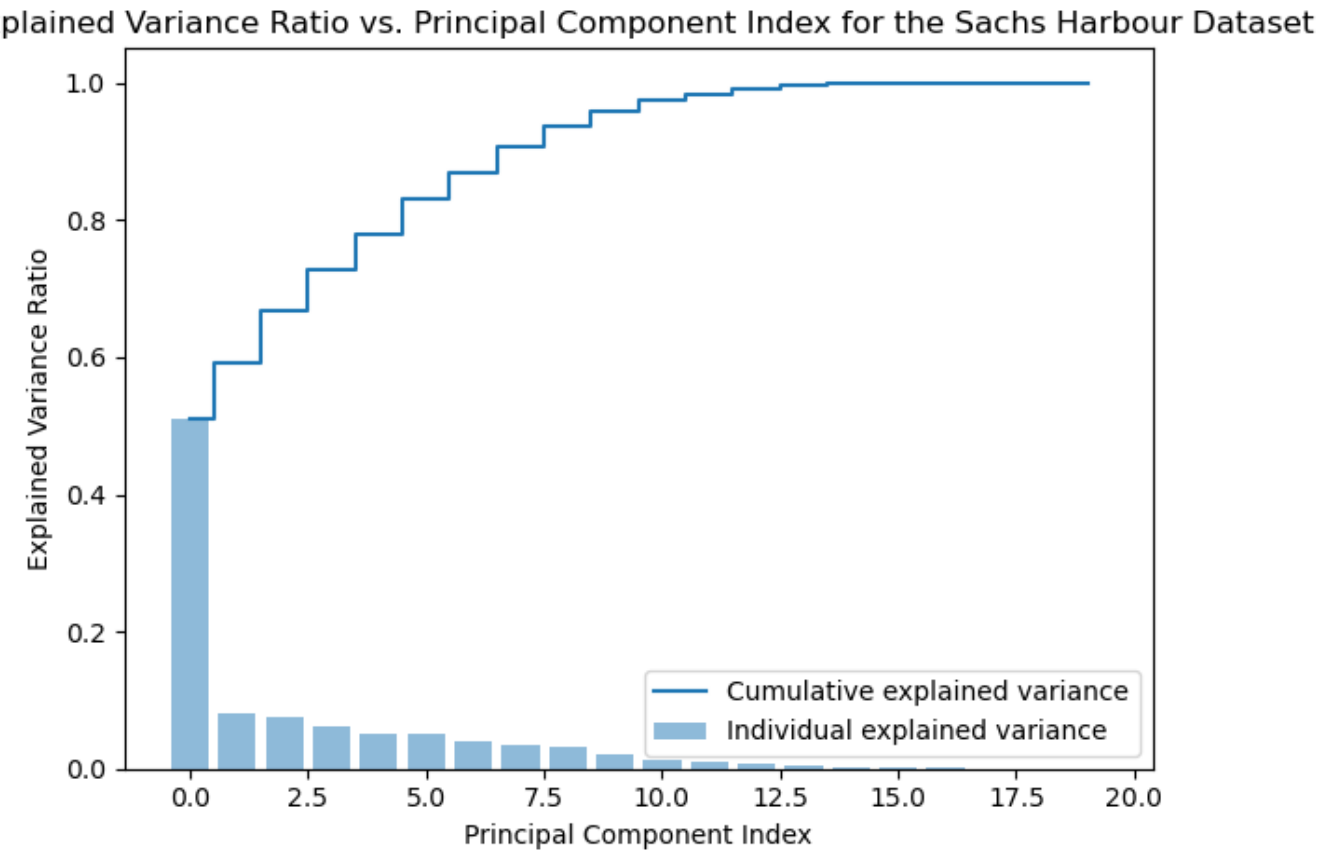
Out[19]:
```
Text(0.5, 0, 'X6')
```



Above is the normalized training and validation data, which span the same range.

From the below PCA analysis, 50% of the total variance is contained within the first principal component. For the sake of this analysis, PCA was not used, as the data did not require too extensive computational resources.

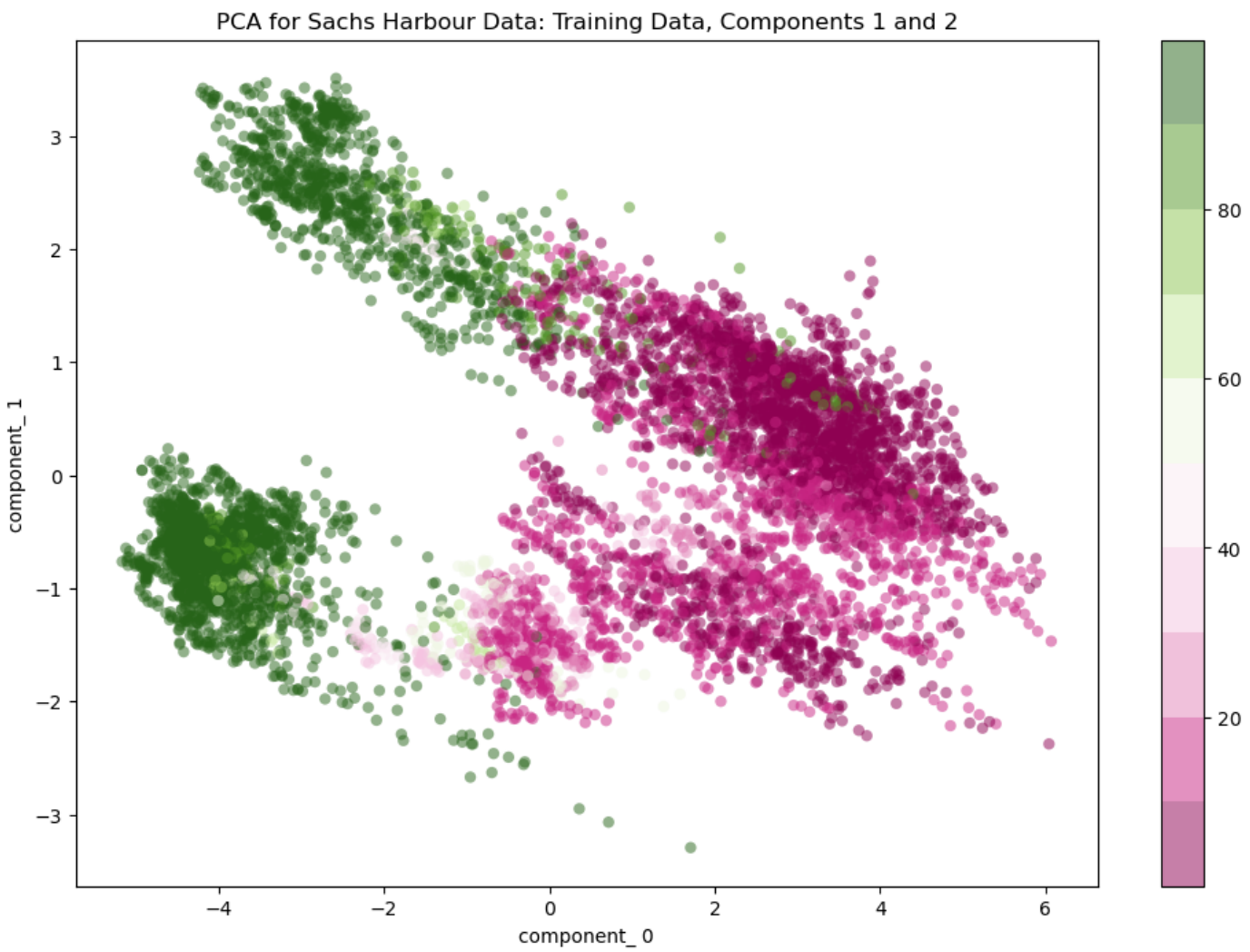
In [20]:
```
PCA_analysis(df_SH)
```



Below is the t-sne analysis for principal components 1 and 2 of the Sachs Harbour data set. The analysis is able to distinguish patterns in the data, meaning that there is a discoverable relationship in the data.

In [21]:
```
n_components=0.95

pca=PCA(n_components=n_components, copy=True, whiten=False, svd_solver='auto', tol=0.0,
pca.fit(X_tr_Norm)
train_pca = pca.transform(X_tr_Norm)
test_pca = pca.transform(X_va_Norm)
comX=0
comY=1

plt.figure(figsize=(12,8))
plt.scatter(train_pca[:,comX], train_pca[:,comY],
            c=Y_tr, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('PiYG',10))
plt.xlabel('component_ '+ str(comX))
plt.ylabel('component_ '+str(comY))
plt.colorbar();
plt.title('PCA for Sachs Harbour Data: Training Data, Components 1 and 2')
```

Out[21]:
```
Text(0.5, 1.0, 'PCA for Sachs Harbour Data: Training Data, Components 1 and 2')
```



ANN

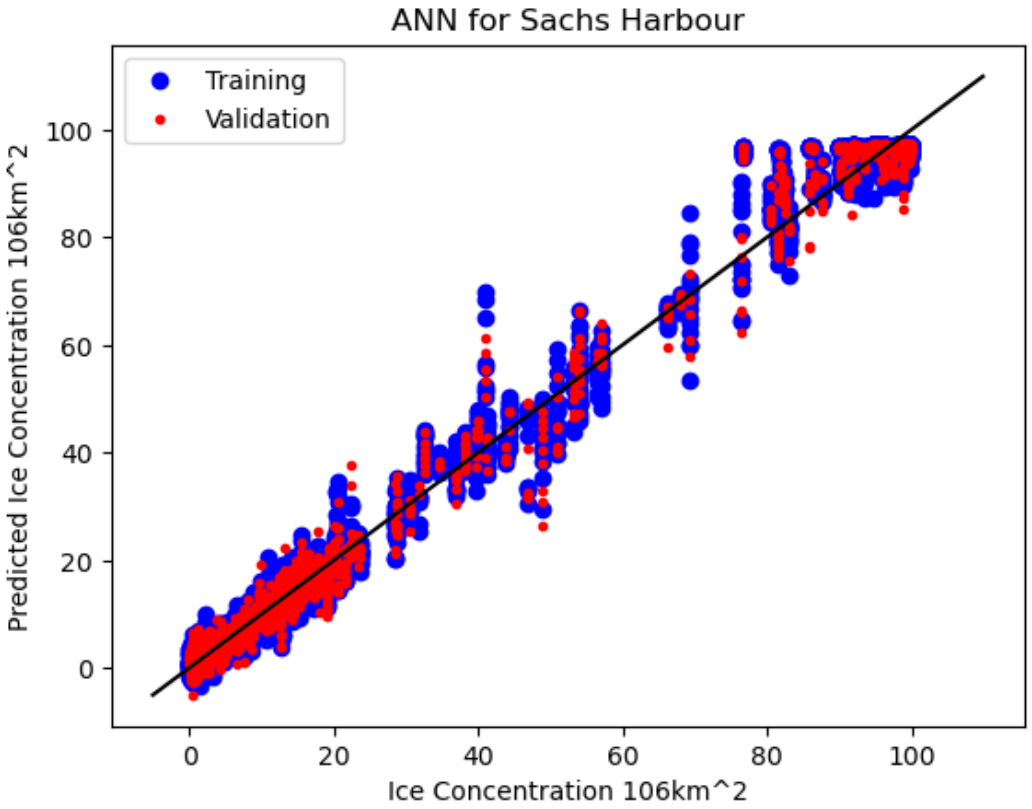
Below is the pipeline I used to optimize the ANN network. Note that some optimized variables have been removed from the last iteration of the pipeline (ie. are no longer in the params section).

In [22]:
```
#ann_pipeline = Pipeline([('ANNreg', MLPRegressor(n_iter_no_change=5,
#                                                  validation_fraction=0.1, early_stoppin
#                                                 learning_rate='adaptive', solver='adam'

#params = [{'ANNreg__activation':['relu', 'tanh'], 'ANNreg__hidden_layer_sizes':[(20,20)
#          'ANNreg__alpha':[0.0001, 1], 'ANNreg__tol':[0.0001, 1]}]

#gs_ann = GridSearchCV(ann_pipeline,
#                      param_grid=params,
#                      scoring='neg_mean_squared_error',
#                      cv=5)

#gs_ann.fit(X_tr_Norm, Y_tr)

#gs_ann.best_params_
```

In [23]:
```
# Initialize the ANN
reg_ann = MLPRegressor(hidden_layer_sizes=(20,20), n_iter_no_change=5,
                       validation_fraction=0.1, early_stopping=True, max_iter=500,
                       activation='tanh', solver='adam', learning_rate='adaptive', alpha

# Fit the ANN on the data
reg_ann.fit(X_tr_Norm, Y_tr)
```

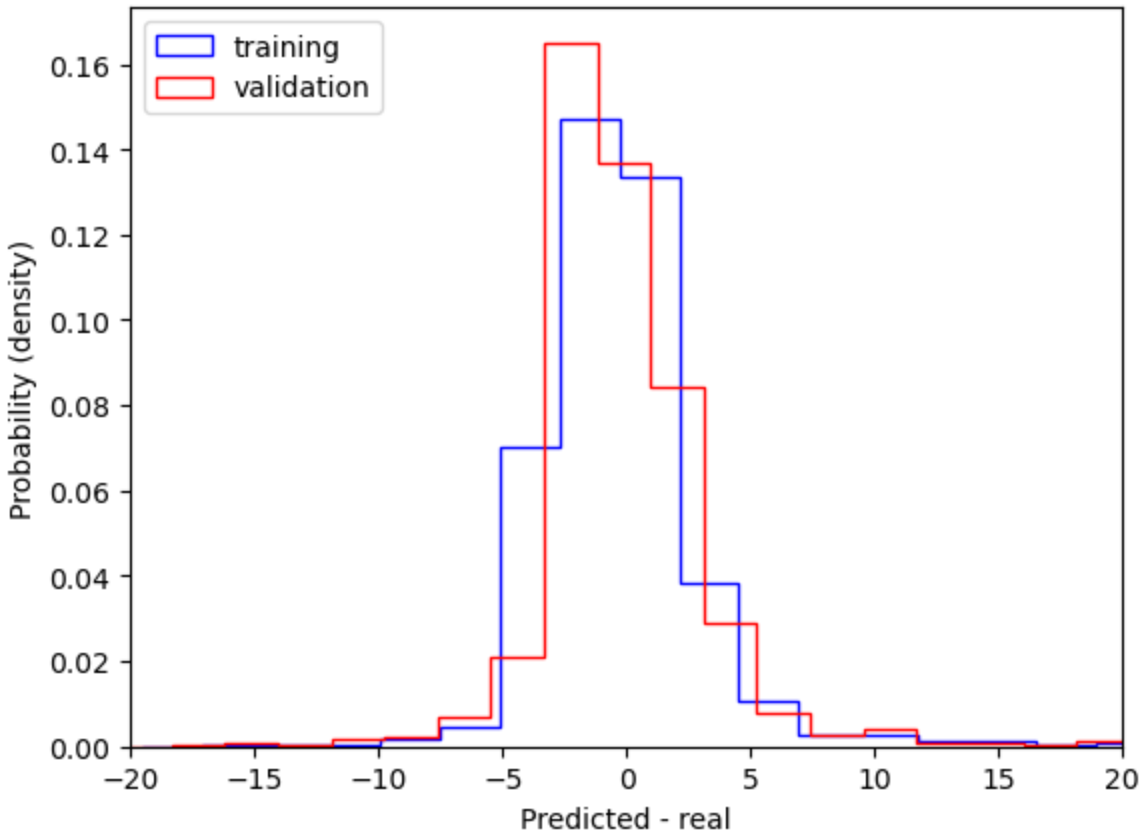
```
# Predict the results
Y_tr_pred_ann = reg_ann.predict(X_tr_Norm)
Y_va_pred_ann = reg_ann.predict(X_va_Norm)
```

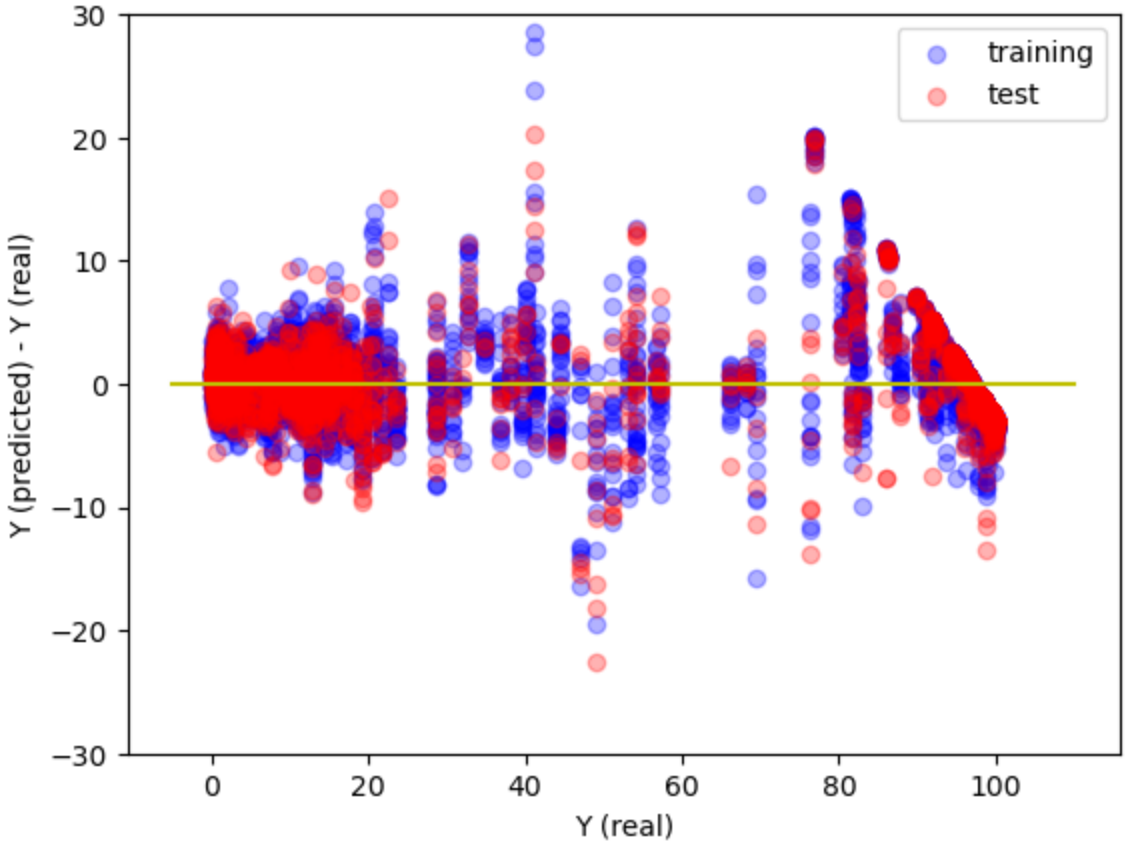
In [24]:
```
# Plot the results
stats_plots(Y_tr, Y_va, Y_tr_pred_ann, Y_va_pred_ann, -5, 110, 125, 10, 125, 40, title='
```



$\mu = 0.3003$
$\text{med} = 0.6050$
$\sigma = 3.2916$

$\mu = 0.2731$
$\text{med} = 0.5637$
$\sigma = 3.0361$





ANN Discussion

The ANN does a good job at predicting the data, with a standard deviation of 3.29 for the validation set and 3.04 for the training set. The first plot shows the relationship between the actual ice concentration value and the predicted ice concentration value, which shows a good linear relationship with some noise, as this is real world collected data. The gaps in the data may be representative of the lack of information in the spring and fall months, as most of the year is either 100% covered in ice or 0% covered in ice, as this station is close to shore.

The second plot shows the probability density, and the training and validation sets show a similar distribution, which is a good sign that the network has accurately learned to classify future predictions.

The third plot shows the error in the training and validation (or test) sets, and this spread is as expected for real world data. The data seems to be centered along 0, meaning that there isn't a shift towards over predicting or underpredicting the data.

These results are all indicative of a well trained model, meaning that I do not expect there is substantial overfitting occuring here.

SVM

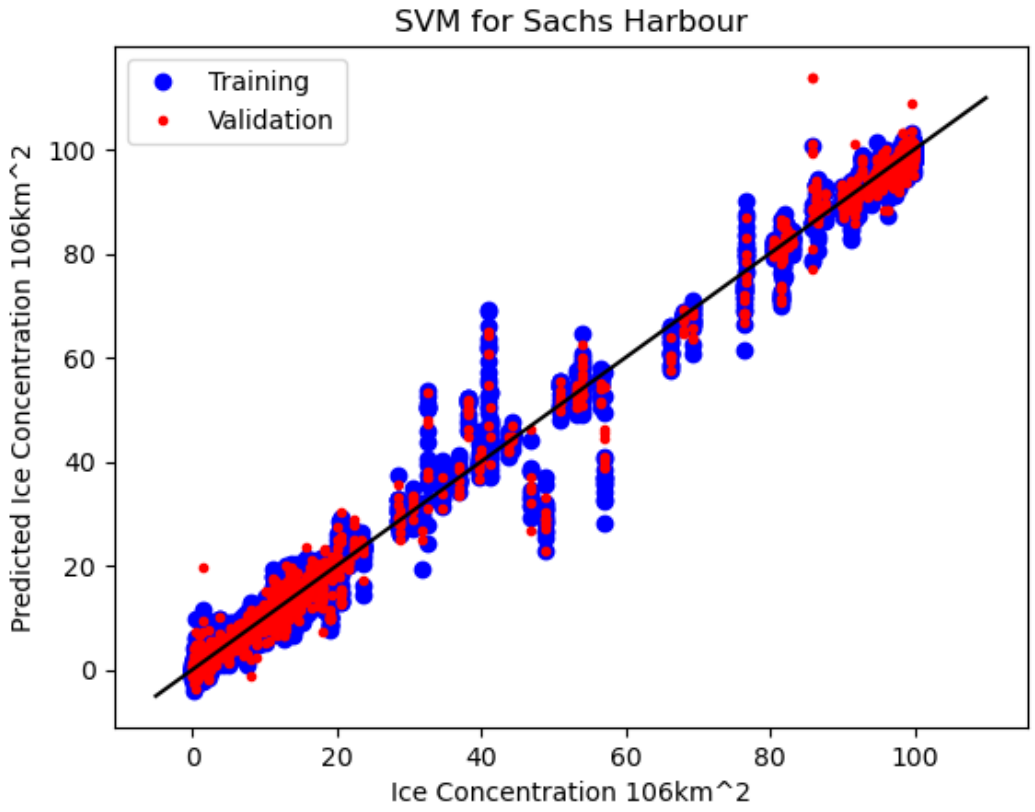
Below is the pipeline I used to optimize the SVM network. Note that some optimized variables have been removed from the last iteration of the pipeline (ie. are no longer in the params section).

In [25]:
```
# I have done multiple pipeline iterations, determining the things below.
#svm_pipeline = Pipeline([('svr', SVR(kernel='poly', degree=3, C=5, coef0=1, epsilon=0.0

#params = [{'svr__gamma':[0.1, 1], 'svr__tol':[0.0001, 0.01], 'svr__epsilon':[0.001, 0.0
```

```
#gs_svm = GridSearchCV(svm_pipeline,
#                       param_grid=params,
#                       scoring='neg_mean_squared_error',
#                       cv=5)

#gs_svm.fit(X_tr_Norm, Y_tr)

#gs_svm.best_params_
```

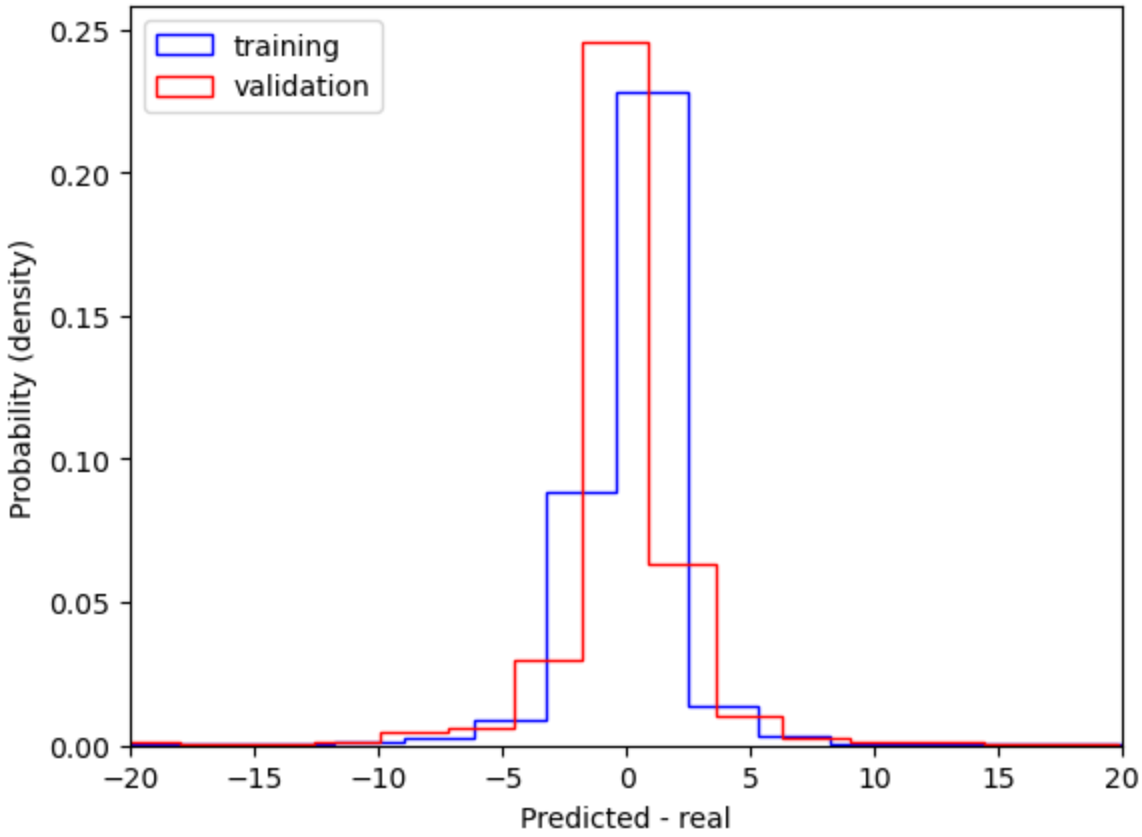
In [26]:
```
# Initialize and fit the model - note: this takes a long time to run, so I would not rec
clf_svm = SVR(kernel='poly', gamma=0.5, C=6, epsilon=0.001, degree=3, coef0=1, tol=0.01)
clf_svm.fit(X_tr_Norm, Y_tr)
Y_tr_pred = clf_svm.predict(X_tr_Norm)
Y_va_pred = clf_svm.predict(X_va_Norm)
```

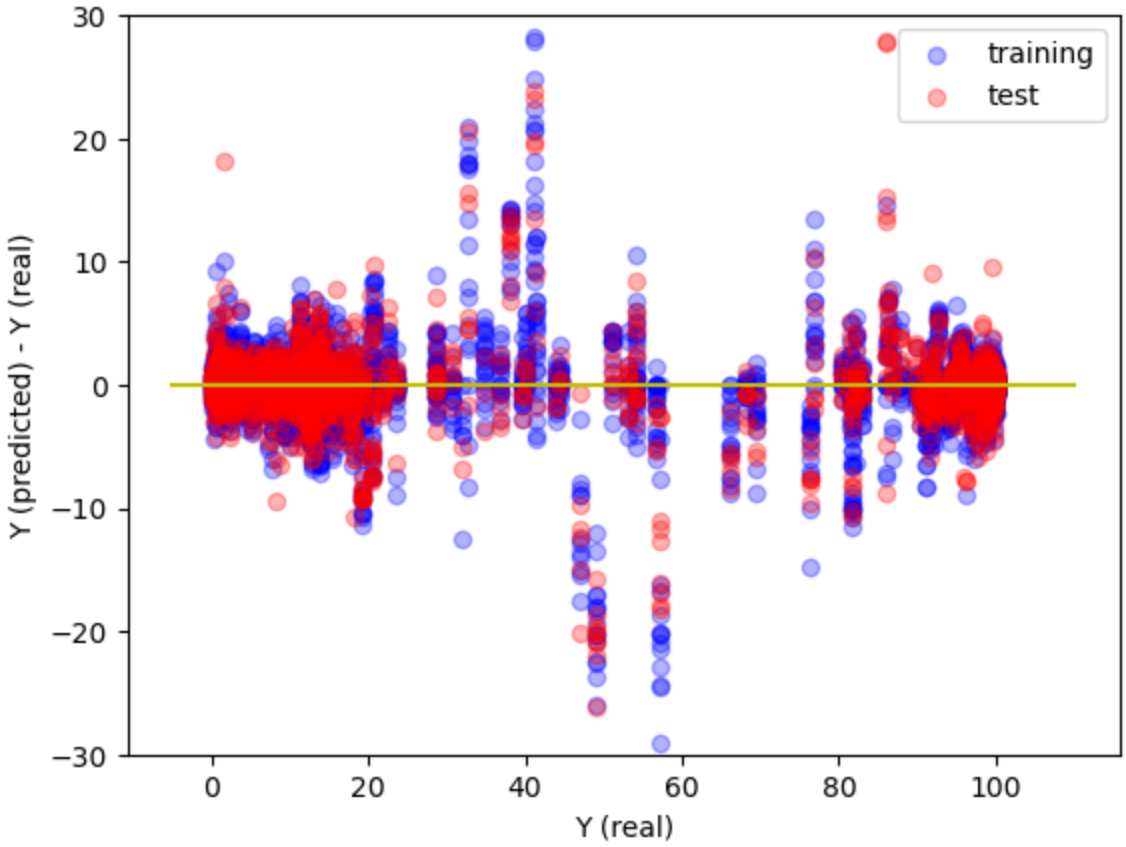
In [27]:
```
# Plot the stats plots
stats_plots(Y_tr, Y_va, Y_tr_pred, Y_va_pred, -5, 110, 125, 10, 125, 40, title='SVM for
```



$\mu = 0.0787$
$\text{med} = 0.0334$
$\sigma = 3.1012$

$\mu = 0.0380$
$\text{med} = 0.0004$
$\sigma = 2.5142$





SVM Discussion

The SVM does a good job at predicting the results, similarly to the ANN as discussed previously. The probability density is more dense at zero for the SVM than the ANN (for both the validation and training sets), meaning that the SVM more accurately predicted more measurements than the ANN. Similar to the ANN, there does not seem to be a shift towards over or under predicting values, as seen in the third plot above.

Conclusion (Regression: ANN vs SVM)

The table below provides a summary of the results:

Stat	ANN (Tr)	ANN (Va)	SVM (Tr)	SVM (Va)
Mean	0.273	0.300	0.038	0.079
Median	0.534	0.605	0.0004	0.033
Std	3.036	3.292	2.514	3.101

The SVM outperformed the ANN for the task of regression for the Sachs Harbour data set. The standard deviation was lower for the SVM, and the number of accurately predicted values higher. For future predictions with this data, I would select the SVM regression algorithm.

Conclusion

SVM has been shown here to be a powerful algorithm, outperforming a 3-layer neural network for both classification and regression tasks.

For the forest cover data set (classification), an almost 10% increase in accuracy occured for the SVM compared to the ANN. For this imbalanced data set, undersampling was used to balance the classes before analysis to ensure that the network could predict each class with no bias for one or the other. To ensure the results were generalizable, the ANN and SVM algorithms were ran 5 times and averages were used to compare performance with associated errors presented from standard deviation. With more time and computational resources, running more iterations would make these results even more generalizable. Interestingly, the data set was given with more samples of class 1 and class 2, and these were the classes that suffered from the ANN analysis. It might be the case that class 1 and class 2 needed more samples to be accurately predicted. It would be interesting to run this analysis again with the imbalanced data and see if the ANN results improve. For future classifications, I would use the SVM algorithm.

For the Sachs Harbour data set (regression), the ANN and SVM performed very similarly, but with the SVM algorithm on top in terms of performance metrics. Either algorithm would be a good choice for this data, but due to run times, I would slightly suggest the ANN for this data if reruns were necessary. The ANN ran in under 2 minutes, whereas the SVM took more than 30. The results between the two algorithms are fairly similar, and so because of the extensive run time, if models needed to be rerun for some reason, I would suggest the ANN.

Thank you, Hossen! This was a great course.

References

Literature:

Bishop, C. M. (2006). Pattern recognition and machine learning. In Pattern recognition and machine learning. Springer.

Noble, W. (2006). What is a support vector machine?. Nat Biotechnol 24, 1565–1567 (2006). https://doi.org/10.1038/nbt1206-1565

Suthaharan, S. (2016). Support Vector Machine. In: Machine Learning Models and Algorithms for Big Data Classification. Integrated Series in Information Systems, vol 36. Springer, Boston, MA. https://doi-org.ezproxy.library.uvic.ca/10.1007/978-1-4899-7641-3_9

Forest Cover Dataset: https://www.kaggle.com/datasets/uciml/forest-cover-type-dataset

Sachs Harbour Dataset: Insley, S. J., Halliday, W. D., & de Jong, T. (2017). Seasonal Patterns in Ocean Ambient Noise near Sachs Harbour, Northwest Territories. Arctic, 70(3), 239–248. https://doi.org/10.14430/arctic4662

Scikit-Learn Documentation:

https://scikit-learn.org/stable/modules/svm.html#svm-regression

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html