**Sarah Riaz**
**USC ID: 1389209718**
**sriaz@usc.edu**

# Homework 5

## Question 1:

### a)

I used twenty percent of each class (0 and 1) of 'Whether' for testing and the remaining data for training.

**Code:**

```
import numpy as np
import pandas as pd

dataframe = pd.read_csv("transfusion.csv")

dataset_for_0 = (dataframe.loc[dataframe['whether'] == 0])
dataset_for_1 = (dataframe.loc[dataframe['whether'] == 1])

predictor_count = 4

training_set_X_for_0 = dataset_for_0.values[(np.int(len(dataset_for_0.values) *
0.20)):, :predictor_count]
training_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values)
* 0.20)), predictor_count:]
training_set_X_for_1 = dataset_for_1.values[(np.int(len(dataset_for_1.values) *
0.20)):, :predictor_count]
training_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values)
* 0.20)), predictor_count:]

testing_set_X_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
0.20)), :predictor_count]
testing_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
0.20)), predictor_count:]
testing_set_X_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
0.20)), :predictor_count]
testing_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
0.20)), predictor_count:]

training_set_X = np.vstack((training_set_X_for_0, training_set_X_for_1))
training_set_Y = np.vstack((training_set_Y_for_0, training_set_Y_for_1))
testing_set_X = np.vstack((testing_set_X_for_0, testing_set_X_for_1))
testing_set_Y = np.vstack((testing_set_Y_for_0, testing_set_Y_for_1))
```

### b) Supervised Learning:

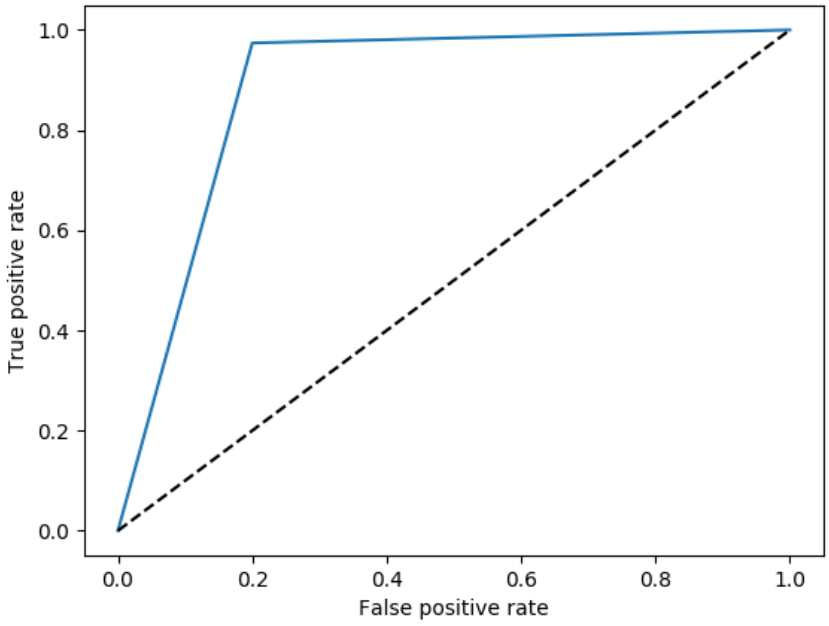
For normalization of data, I have used sklearn's preprocessing package and for imbalance reduction I have used SMOTE so that the minority class can be over-sampled. Then, I have used five fold cross-validation to find the best value for L1 penalty parameter and then, to get the best model for supervised classification, I have used GridSearchCV.

**Results:**

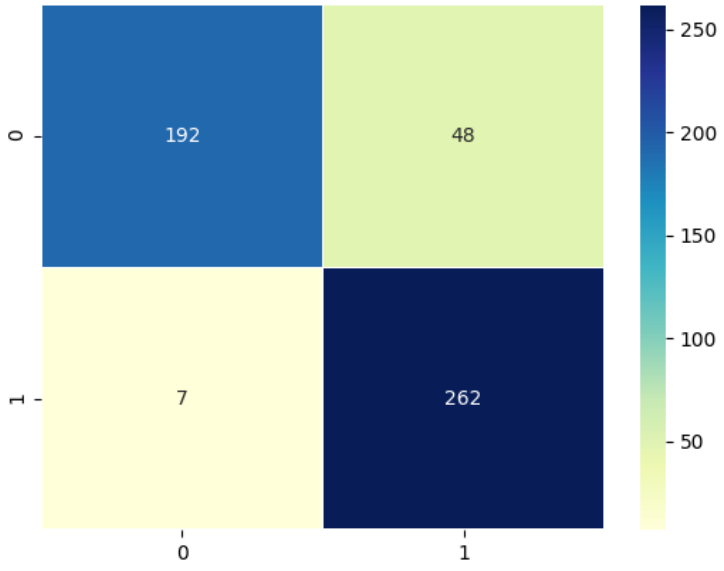
**Training Set:**
Best SVM Penalty Param = C = 0.966039211809
Best Score using C = 0.86740471823411

Prediction Accuracy of Training Dataset = 0.892819324421
Area under the Curve of Training Dataset = 0.896977847584

ROC Curve with AUC = 0.896977847584

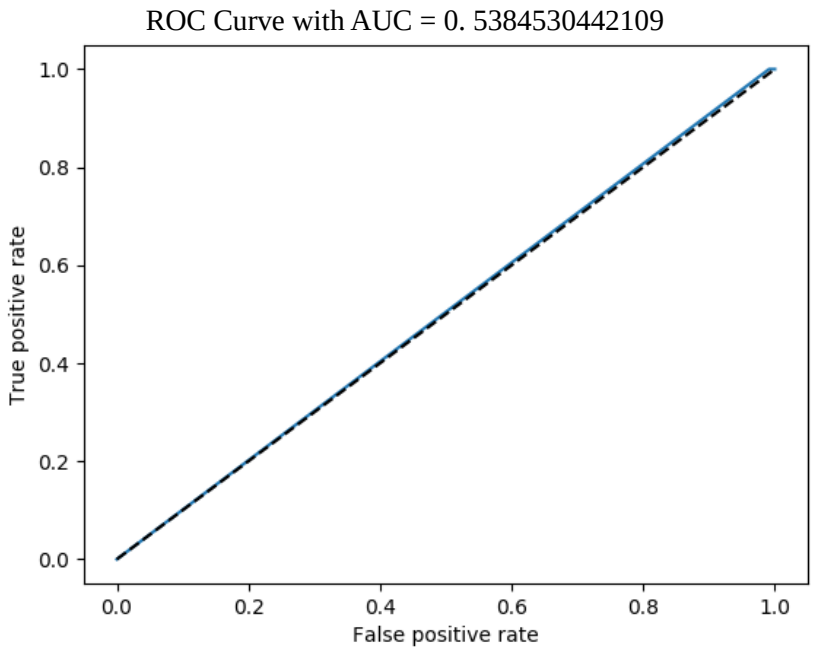


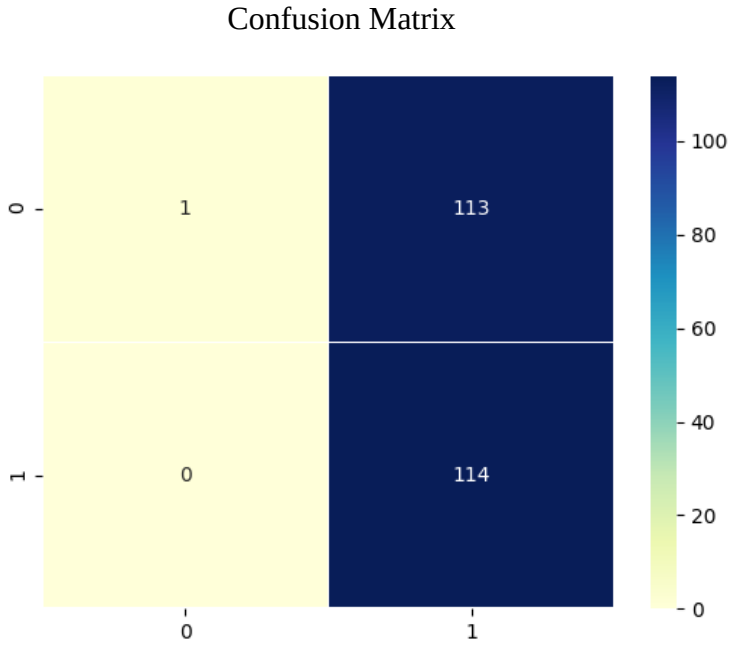

Confusion Matrix




**Testing Set:**

Prediction Accuracy of Testing Dataset = 0.5384530442109
Area under the Curve of Testing Dataset = 0.5384530442109




As it can be observed in the results, ROC and confusion matric that the classifier could not classify class 0 successfully which resulted in a large number of misclassifications.

## Code:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import normalize
from sklearn.svm import LinearSVC
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
from sklearn.utils import shuffle
from imblearn.combine import SMOTEENN, SMOTETomek
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix
from sklearn.metrics import auc
from sklearn.metrics import mean_squared_error
from scipy.sparse import coo_matrix, hstack
import seaborn as sns

dataframe = pd.read_csv("transfusion.csv")

dataset_for_0 = (dataframe.loc[dataframe['whether'] == 0])
dataset_for_1 = (dataframe.loc[dataframe['whether'] == 1])

predictor_count = 4

training_set_X_for_0 = dataset_for_0.values[(np.int(len(dataset_for_0.values) * 0.20)):,
:predictor_count]
training_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) * 0.20)),
predictor_count:]
training_set_X_for_1 = dataset_for_1.values[(np.int(len(dataset_for_1.values) * 0.20)):,
:predictor_count]
training_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) * 0.20)),
predictor_count:]

testing_set_X_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) * 0.20)),
:predictor_count]
testing_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) * 0.20)),
predictor_count:]
testing_set_X_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) * 0.20)),
:predictor_count]
testing_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) * 0.20)),
predictor_count:]

training_X = normalize(np.vstack((training_set_X_for_0, training_set_X_for_1)),
norm='l2')
training_Y = np.vstack((training_set_Y_for_0, training_set_Y_for_1))
testing_X = normalize(np.vstack((testing_set_X_for_0, testing_set_X_for_1)), norm='l2')
testing_Y = np.vstack((testing_set_Y_for_0, testing_set_Y_for_1))

training_X_new = shuffle(hstack([coo_matrix(training_X),
coo_matrix(training_Y)]).toarray(), random_state=15)
testing_X_new = shuffle(hstack([coo_matrix(testing_X),
coo_matrix(testing_Y)]).toarray(), random_state=36)

training_X = training_X_new[:, :predictor_count]
training_Y = training_X_new[:, predictor_count:]
testing_X = testing_X_new[:, :predictor_count]
testing_Y = testing_X_new[:, predictor_count:]

sampling = SMOTEENN(random_state=5,kind_smote='svm')
training_smote_X, training_smote_Y = sampling.fit_sample(training_X, training_Y.ravel())
testing_smote_X, testing_smote_Y = SMOTE().fit_sample(testing_X, testing_Y.ravel())
testing_smote_Y=testing_smote_Y.reshape(len(testing_smote_Y), 1)

testing_X_new = shuffle(hstack([coo_matrix(testing_smote_X),
coo_matrix(testing_smote_Y)]).toarray(), random_state=65)
testing_X = testing_X_new[:, :predictor_count]
testing_Y = testing_X_new[:, predictor_count:]

classifier = GridSearchCV(LinearSVC(penalty='l1', dual=False, tol=0.001), [{'C':
np.linspace(0.0001, 1, 250)}], cv=KFold(5), refit=True, n_jobs=4)
```

```
classifier.fit(training_smote_X, training_smote_Y.ravel())

predictions = classifier.predict(training_smote_X)
accuracy = 1 - mean_squared_error(training_smote_Y, predictions)
best_SVM_param = classifier.best_params_
best_score = classifier.best_score_

test_set_predictions = classifier.predict(testing_X)
accuracy = 1 - mean_squared_error(testing_Y, test_set_predictions)


fpr_rf_lm, tpr_rf_lm, _ = roc_curve(training_smote_Y, predictions, pos_label=1)
roc_auc = auc(fpr_rf_lm, tpr_rf_lm)
plt.plot(fpr_rf_lm, tpr_rf_lm, label=str(roc_auc))
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.plot([0, 1], [0, 1], 'k--')
plt.show()

fpr_rf_lm, tpr_rf_lm, _ = roc_curve(testing_Y, test_set_predictions, pos_label=1)
roc_auc = auc(fpr_rf_lm, tpr_rf_lm)
plt.plot(fpr_rf_lm, tpr_rf_lm, label=str(roc_auc))
plt.title('ROC curve- (Test-Set), AUC: '+str(roc_auc))
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.plot([0, 1], [0, 1], 'k--')
plt.show()

confusion_matrix=confusion_matrix(training_smote_Y, predictions)
sns.heatmap(confusion_matrix, cmap="Blues", annot=True, linewidths=0.5, fmt='d')
plt.show()

confusion_matrix = confusion_matrix(testing_Y, test_set_predictions)
sns.heatmap(confusion_matrix, cmap="Blues", annot=True, linewidths=0.5, fmt='d')
plt.show()
```

## c) Semi-Supervised Learning

i) For semi-supervised learning, I have used Linear SVM with L1 penalty and the labeled training data was used to fit the model. Then, I used GridSearchCV to perform five fold cross-validation on this model.

ii) Then, to find the unlabeled data point closet to the decision boundary I used the decision function of LinearSVC.

iii) Then, I added this data point labeled by the model to the labeled set of data and removed it from the unlabeled set of data.

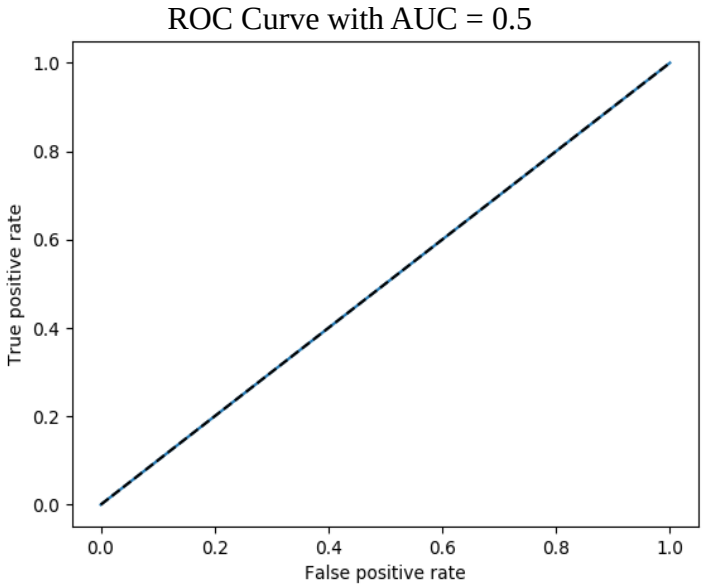
iv) Then, I calculated the misclassification rate and accuracy of the predictions made by the model on test dataset.

**Results:**

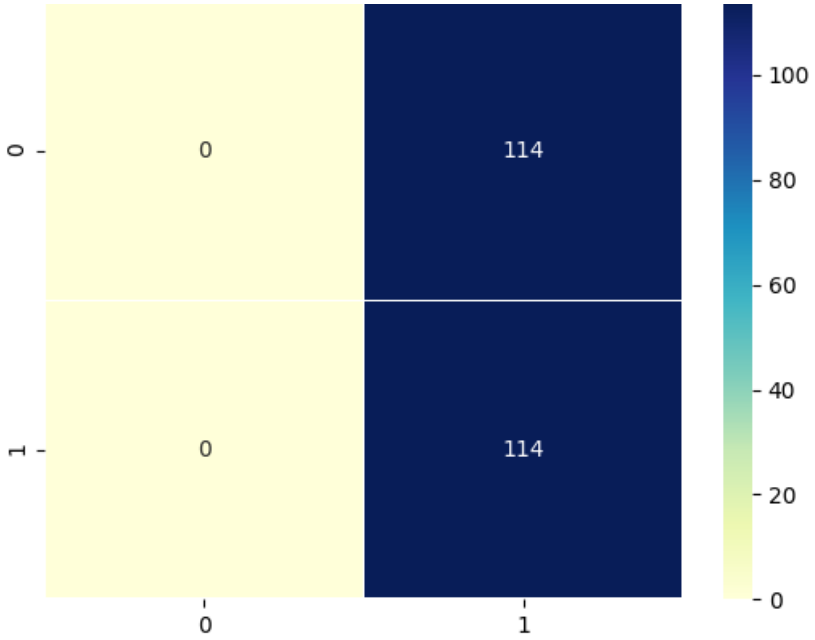
**Training:**
Best SVM Penalty Param for index 0 = C = 8.9891282324
Best Score using C for index 0 = 0.932497317
Best SVM Penalty Param for index 216 = C = 6.293474182

Best Score using C for index 216 = 0.8413402377

**Testing:**
Area Under the Curve = 0.5
Accuracy = 0.5




Confusion Matrix




Results show that semi-supervised learning fails at predicting the test dataset correctly. The number of misclassifications is huge—an entire class is misclassified.

**Code:**

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import normalize
from sklearn.svm import LinearSVC
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
from sklearn.utils import shuffle
from imblearn.combine import SMOTEENN, SMOTETomek
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix
from sklearn.metrics import auc
from sklearn.metrics import mean_squared_error
from scipy.sparse import coo_matrix, hstack
import seaborn as sns

dataframe = pd.read_csv("transfusion.csv")

dataset_for_0 = (dataframe.loc[dataframe['whether'] == 0])
dataset_for_1 = (dataframe.loc[dataframe['whether'] == 1])

predictor_count = 4

percentage = 0.20
training_set_X_for_0 = dataset_for_0.values[(np.int(len(dataset_for_0.values) *
percentage)):, :predictor_count]
training_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), predictor_count:]
training_set_X_for_1 = dataset_for_1.values[(np.int(len(dataset_for_1.values) *
percentage)):, :predictor_count]
training_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), predictor_count:]

testing_set_X_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), :predictor_count]
testing_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), predictor_count:]
testing_set_X_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), :predictor_count]
testing_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), predictor_count:]

testing_set_for_Y = np.vstack((testing_set_Y_for_0, testing_set_Y_for_1))
testing_set_for_X = normalize(np.vstack((testing_set_X_for_0, testing_set_X_for_1)),
norm='l2')

training_set_X_for_0_half = int(len(training_set_X_for_0) * 0.50)
training_set_X_for_1_half = int(len(training_set_X_for_1) * 0.50)

labelled_training_set_X_for_0 = training_set_X_for_0[training_set_X_for_0_half:, :]
labelled_training_set_for_Y_for_0 = training_set_Y_for_0[training_set_X_for_0_half:, :]
unlabelled_training_set_X_for_0 = training_set_X_for_0[:training_set_X_for_0_half, :]
unlabelled_training_set_Y_for_0 = training_set_Y_for_0[:training_set_X_for_0_half, :]

labelled_training_set_X_for_1 = training_set_X_for_1[training_set_X_for_1_half:, :]
labelled_training_set_Y_for_1 = training_set_Y_for_1[training_set_X_for_1_half:, :]
unlabelled_training_set_X_for_1 = training_set_X_for_1[:training_set_X_for_1_half, :]
unlabelled_training_set_Y_for_1 = training_set_Y_for_1[:training_set_X_for_1_half, :]

labelled_training_set_for_Y = np.vstack((labelled_training_set_for_Y_for_0,
labelled_training_set_Y_for_1))
unlabelled_training_set_for_Y = np.vstack((unlabelled_training_set_Y_for_0,
unlabelled_training_set_Y_for_1))

unlabelled_training_set_for_X = normalize(np.vstack((unlabelled_training_set_X_for_0,
```

```
unlabelled_training_set_X_for_1)), norm='l2')
X_combined_train_unlabelled = shuffle(hstack([coo_matrix(unlabelled_training_set_for_X),
coo_matrix(unlabelled_training_set_for_Y)]).toarray(), random_state=10)

unlabelled_training_set_for_X = X_combined_train_unlabelled[:, :predictor_count]
unlabelled_training_set_for_Y = X_combined_train_unlabelled[:, predictor_count:]

sampling=SMOTEENN(random_state=5, kind_smote='svm')
X_train_unlabelled_smote, Y_train_unlabelled_smote =
sampling.fit_sample(unlabelled_training_set_for_X,
unlabelled_training_set_for_Y.ravel())

X_train_labelled=normalize(np.vstack((labelled_training_set_X_for_0,
labelled_training_set_X_for_1)), norm='l2')

index_to_use = hstack([coo_matrix(X_train_labelled),
coo_matrix(labelled_training_set_for_Y)]).toarray()
X_combined_train_labelled = shuffle(index_to_use, random_state=10)

X_train_labelled=X_combined_train_labelled[:, :predictor_count]
labelled_training_set_for_Y= X_combined_train_labelled[:, predictor_count:]

sampling=SMOTEENN(random_state=5, kind_smote='svm')
X_train_labelled_smote, Y_train_labelled_smote = sampling.fit_sample(X_train_labelled,
labelled_training_set_for_Y.ravel())

X_combined_test = shuffle(hstack([coo_matrix(testing_set_for_X),
coo_matrix(testing_set_for_Y)]).toarray(), random_state=35)


testing_set_for_X= X_combined_test[:, :predictor_count]
testing_set_for_Y= X_combined_test[:, predictor_count:]

X_test_smote, Y_test_smote = SMOTE().fit_sample(testing_set_for_X,
testing_set_for_Y.ravel())
Y_test_smote=Y_test_smote.reshape(len(Y_test_smote),1)

X_combined_test = shuffle(hstack([coo_matrix(X_test_smote),
coo_matrix(Y_test_smote)]).toarray(), random_state=35)

testing_set_for_X= X_combined_test[:, :predictor_count]
testing_set_for_Y= X_combined_test[:, predictor_count:]

training_iterations = len(X_train_unlabelled_smote)
for i in range(1, training_iterations):
    classifier = GridSearchCV(LinearSVC(penalty='l1', dual=False, tol=0.001), [{'C':
np.linspace(0.0001, 10, 50)}], cv=KFold(5), refit=True, n_jobs=4)
    classifier.fit(X_train_labelled_smote, Y_train_labelled_smote.ravel())

    scores = classifier.cv_results_['mean_test_score']
    scores_std = classifier.cv_results_['std_test_score']

    best_SVM_param = classifier.best_params_
    best_score = classifier.best_score_

    X_margin = np.abs(classifier.decision_function(X_train_unlabelled_smote))
    dataset = pd.DataFrame(data={'index': np.arange(0, len(X_margin), 1),
'distance_of_margin': X_margin})
    dataset = dataset.sort_values(by='distance_of_margin')

    check_arr = X_train_labelled_smote
    check_Y_arr = Y_train_labelled_smote

    to_use_indices = []
    top_data = pd.DataFrame()
    top_data = dataset.iloc[:1]
```

```
    for j in range(0, 1):
        index_to_use = np.int(top_data.values[j, 1])
        to_use_indices.append(index_to_use)
        check_arr = np.vstack((check_arr,
np.array(X_train_unlabelled_smote[index_to_use].reshape(1, 4))))
        Y_train_unlabelled_smote[index_to_use] =
classifier.predict(X_train_unlabelled_smote[index_to_use:index_to_use + 1, :])
        check_Y_arr = np.vstack((check_Y_arr.reshape(len(check_Y_arr),1),
(Y_train_unlabelled_smote[index_to_use].reshape(1, 1))))

    X_train_unlabelled_smote = np.delete(X_train_unlabelled_smote, to_use_indices,
axis=0)
    Y_train_unlabelled_smote = np.delete(Y_train_unlabelled_smote, to_use_indices,
axis=0)

    X_train_labelled_smote = check_arr
    Y_train_labelled_smote = check_Y_arr

    if i == training_iterations - 1:
        test_set_predictions = classifier.predict(testing_set_for_X)
        accuracy = 1 - mean_squared_error(testing_set_for_Y, test_set_predictions)

        fpr_rf_lm, tpr_rf_lm, _ = roc_curve(testing_set_for_Y, test_set_predictions,
pos_label=1)
        roc_auc = auc(fpr_rf_lm, tpr_rf_lm)
        plt.plot(fpr_rf_lm, tpr_rf_lm, label=str(roc_auc))
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.plot([0, 1], [0, 1], 'k--')
        plt.show()

        confusion_matrix = confusion_matrix(testing_set_for_Y, test_set_predictions)
        sns.heatmap(confusion_matrix, cmap="Blues", annot=True, linewidths=0.5, fmt='d')
        plt.show()
```

**d) Unsupervised Learning:**

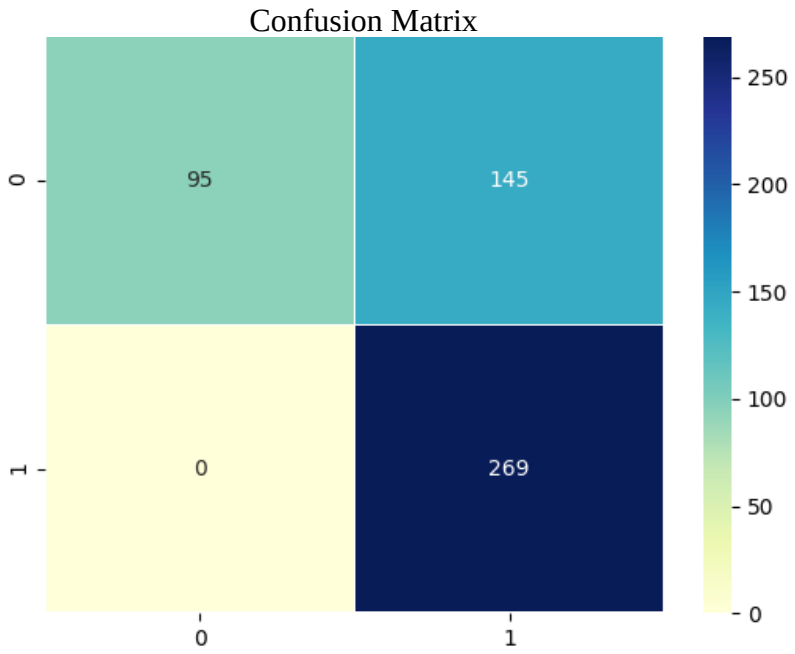
i) To avoid getting into a local minima, I used hyper parameter *n_init* to run the k-means algorithm hundred times and then, out of all the different values of number of clusters best value is chosen which has the least distortion**.**

ii) The parameter **Kmeans.cluster_centers_** gives the co-ordinates of the center for each of the clusters. For the Euclidean distance of each of the data points from the center of each cluster, I used *kmeans.transform()* which transformed the dataset into [Number_of_samples X Number_of_clusters] matrix. Each data point falls into the cluster it has minimum Euclidean distance from its center. To get thirty data points closet to each cluster, I sorted this matrix. Then, to label each cluster with either 0 or 1, I used majority polling.

iii) To calculate the misclassification rate and accuracy of the model, I used the k-mean model to predict the labels of the data points and compared them to their true labels.

**Results:**

Centroid of first cluster = [0.06910311 0.00396899 0.99224684 0.08623242]
Centroid of second cluster = [0.00615781 0.00399807 0.99951781 0.02551618]
Accuracy = 0.7151277013752455




## Code:

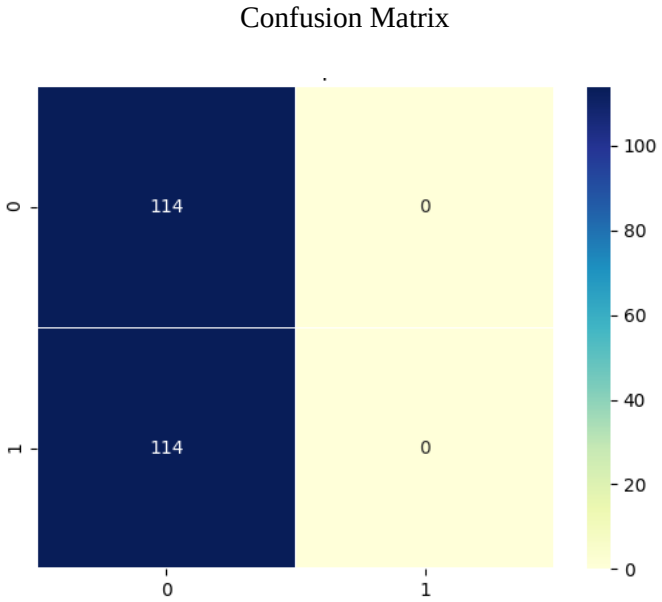
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import normalize
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from imblearn.combine import SMOTEENN
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix
from scipy.sparse import coo_matrix, hstack
import seaborn as sns

dataframe = pd.read_csv("transfusion.csv")

dataset_for_0 = (dataframe.loc[dataframe['whether'] == 0])
dataset_for_1 = (dataframe.loc[dataframe['whether'] == 1])

predictor_count = 4

percentage = 0.20
training_set_X_for_0 = dataset_for_0.values[(np.int(len(dataset_for_0.values) *
percentage)):, :predictor_count]
training_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), predictor_count:]
training_set_X_for_1 = dataset_for_1.values[(np.int(len(dataset_for_1.values) *
percentage)):, :predictor_count]
training_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), predictor_count:]

testing_set_X_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), :predictor_count]
testing_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), predictor_count:]
testing_set_X_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), :predictor_count]
testing_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), predictor_count:]
```

```
training_data_X = normalize(np.vstack((training_set_X_for_0, training_set_X_for_1)),
norm='l2')
training_data_Y = np.vstack((training_set_Y_for_0, training_set_Y_for_1))
testing_data_X = normalize(np.vstack((testing_set_X_for_0, testing_set_X_for_1)),
norm='l2')
testing_data_Y = np.vstack((testing_set_Y_for_0, testing_set_Y_for_1))
training_data_X_all = shuffle(hstack([coo_matrix(training_data_X),
coo_matrix(training_data_Y)]).toarray(), random_state=1)

training_data_X= training_data_X_all[:, :predictor_count]
training_data_Y= training_data_X_all[:, predictor_count:]
training_data_X, training_data_Y = SMOTEENN(random_state = 5,
kind_smote='svm').fit_sample(training_data_X, training_data_Y.ravel())
training_data_Y=training_data_Y.reshape(len(training_data_Y), 1)
testing_data_X_all = shuffle(hstack([coo_matrix(testing_data_X),
coo_matrix(testing_data_Y)]).toarray(), random_state=5)

testing_data_X= testing_data_X_all[:, :predictor_count]
testing_data_Y= testing_data_X_all[:, predictor_count:]

X_test_smote, Y_test_smote = SMOTE().fit_sample(testing_data_X,
testing_data_Y.ravel())
Y_test_smote=Y_test_smote.reshape(len(Y_test_smote),1)
testing_data_X_all = shuffle(hstack([coo_matrix(X_test_smote),
coo_matrix(Y_test_smote)]).toarray(), random_state=5)

testing_data_X = testing_data_X_all[:, :predictor_count]
testing_data_Y = testing_data_X_all[:, predictor_count:]

model = KMeans(n_clusters=2, n_init=20, random_state=5)
model.fit(training_data_X)
inertia = model.inertia_
centroids = model.cluster_centers_
labels = model.labels_
labels = labels.reshape(len(labels), 1)

training_data_X_new = model.transform(training_data_X)
data = {'index': np.arange(0, len(training_data_X_new), 1),
'distance_of_margin_for_first_cluster': training_data_X_new[:, :1].reshape(-1),
'distance_of_margin_for_second_cluster': training_data_X_new[:, 1:2].reshape(-1)}
data_frame = pd.DataFrame(data=data)
top_30_for_first_cluster =
(data_frame.sort_values(by='distance_of_margin_for_first_cluster')).iloc[:30]
top_30_for_second_cluster =
(data_frame.sort_values(by='distance_of_margin_for_second_cluster')).iloc[:30]

zero_counter_for_first_cluster = 0
one_counter_for_first_cluster = 0
zero_counter_for_second_cluster = 0
one_counter_for_second_cluster = 0

for i in range(0, 30):
    label = int(training_data_Y[int(top_30_for_first_cluster.values[i, 2]), 0])
    if label == 1:
        one_counter_for_first_cluster = one_counter_for_first_cluster+ 1
    else:
        zero_counter_for_first_cluster = zero_counter_for_first_cluster + 1

    label = int(training_data_Y[int(top_30_for_second_cluster.values[i, 2]), 0])
    if label == 1:
        one_counter_for_second_cluster = one_counter_for_second_cluster + 1
    else:
        zero_counter_for_second_cluster = zero_counter_for_second_cluster + 1

if one_counter_for_first_cluster >= zero_counter_for_first_cluster:
    class_of_first_cluster = 1
```

```
else:
    class_of_first_cluster = 0

if one_counter_for_second_cluster >= zero_counter_for_second_cluster:
    class_of_second_cluster = 1
else:
    class_of_second_cluster = 0

no_of_missclassifications = 0
for i in range(0, len(labels)):
    if int(labels[i, 0]) != int(training_data_Y[i, 0]):
        no_of_missclassifications = no_of_missclassifications + 1

error = no_of_missclassifications / len(labels)
accuracy = 1 - error
confusion_grid=confusion_matrix(training_data_Y, labels)
sns.heatmap(confusion_grid,cmap="Blues",annot=True, linewidths=0.6,fmt='d')
plt.show()
```

iii. Following results were obtained after predicting the testing dataset on this model:

Accuracy = 0.5




This approach has misclassified class 1, similar to semi-supervised learning technique, therefore, it's performance is unsatisfactory.

**Code:**

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import normalize
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from imblearn.combine import SMOTEENN
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix
from scipy.sparse import coo_matrix, hstack
import seaborn as sns
```

```
dataframe = pd.read_csv("transfusion.csv")

dataset_for_0 = (dataframe.loc[dataframe['whether'] == 0])
dataset_for_1 = (dataframe.loc[dataframe['whether'] == 1])

predictor_count = 4

percentage = 0.20
training_set_X_for_0 = dataset_for_0.values[(np.int(len(dataset_for_0.values) *
percentage)):, :predictor_count]
training_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), predictor_count:]
training_set_X_for_1 = dataset_for_1.values[(np.int(len(dataset_for_1.values) *
percentage)):, :predictor_count]
training_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), predictor_count:]

testing_set_X_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), :predictor_count]
testing_set_Y_for_0 = dataset_for_0.values[:(np.int(len(dataset_for_0.values) *
percentage)), predictor_count:]
testing_set_X_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), :predictor_count]
testing_set_Y_for_1 = dataset_for_1.values[:(np.int(len(dataset_for_1.values) *
percentage)), predictor_count:]

training_data_X = normalize(np.vstack((training_set_X_for_0, training_set_X_for_1)),
norm='l2')
training_data_Y = np.vstack((training_set_Y_for_0, training_set_Y_for_1))
testing_data_X = normalize(np.vstack((testing_set_X_for_0, testing_set_X_for_1)),
norm='l2')
testing_data_Y = np.vstack((testing_set_Y_for_0, testing_set_Y_for_1))
training_data_X_all = shuffle(hstack([coo_matrix(training_data_X),
coo_matrix(training_data_Y)]).toarray(), random_state=1)

training_data_X= training_data_X_all[:, :predictor_count]
training_data_Y= training_data_X_all[:, predictor_count:]
training_data_X, training_data_Y = SMOTEENN(random_state = 5,
kind_smote='svm').fit_sample(training_data_X, training_data_Y.ravel())
training_data_Y=training_data_Y.reshape(len(training_data_Y), 1)
testing_data_X_all = shuffle(hstack([coo_matrix(testing_data_X),
coo_matrix(testing_data_Y)]).toarray(), random_state=5)

testing_data_X= testing_data_X_all[:, :predictor_count]
testing_data_Y= testing_data_X_all[:, predictor_count:]

X_test_smote, Y_test_smote = SMOTE().fit_sample(testing_data_X, testing_data_Y.ravel())
Y_test_smote=Y_test_smote.reshape(len(Y_test_smote),1)
testing_data_X_all = shuffle(hstack([coo_matrix(X_test_smote),
coo_matrix(Y_test_smote)]).toarray(), random_state=5)

testing_data_X = testing_data_X_all[:, :predictor_count]
testing_data_Y = testing_data_X_all[:, predictor_count:]


model = KMeans(n_clusters=2, n_init=20, random_state=5)
model.fit(training_data_X)
inertia = model.inertia_
centroids = model.cluster_centers_
predictions=model.predict(testing_data_X)
predictions=predictions.reshape(len(predictions), 1)

no_of_missclassifications = 0
for i in range(0, len(predictions)):
```

```
        if int(predictions[i, 0]) != int(testing_data_Y[i, 0]):
            no_of_missclassifications = no_of_missclassifications + 1

    error = no_of_missclassifications / len(predictions)
    accuracy = 1 - error
    confusion_grid=confusion_matrix(testing_data_Y, predictions)
    sns.heatmap(confusion_grid,cmap="Blues",annot=True,linewidths=0.6, fmt='d')
    plt.show()
```

**e) Comparison:**

The results show that the as compared to unsupervised learning and semi-supervised learning, supervised learning has performed best in classifying this dataset because the size of dataset is small. Even the semi-supervised learning technique works better than the unsupervised learning.

This can be concluded from the result that on small datasets supervised learning works best whereas unsupervised learning should be used on large datasets.

On our given dataset, supervised learning gives the best results.

## Question 2:

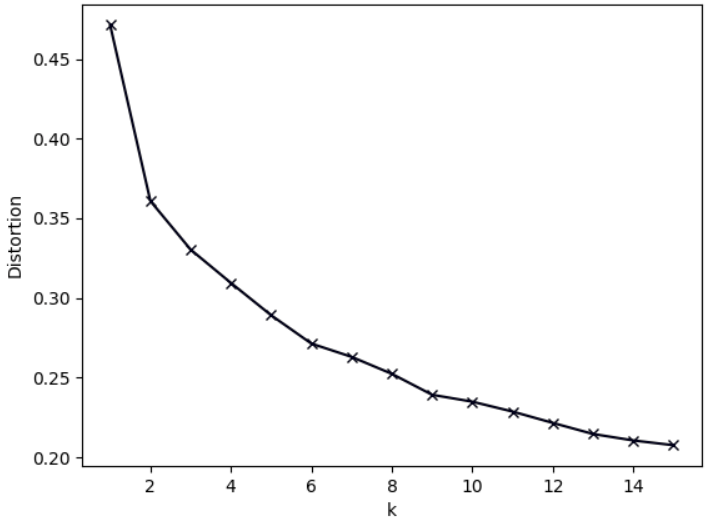
**a) Optimal value of k:**

I have used Elbow method to find the optimal number of clusters to made for the given dataset, which determines the best value of *k* by examining the variance percentage.
In this method, we run k-means clustering on the dataset for a range of values of *k* and for each value of *k* calculate the sum of squared errors (SSE). The optimal value of *k* is selected when an increment of another cluster doesn't give a significant improvement in our model, but if adding another cluster reduces the variance significantly, it will be the optimal value of *k* (if a bigger value doesn't give a significantly better result by reducing the variance). Therefore, when the improvement significantly decreases, we get a pointy graph resulting in an elbow shape.

On our given dataset, using a range of different values (as shown below), the optimal value of *k* is 2 because the most drastic improvement/reduction in variance or distortion is observed with *k* = 2, hence elbow can be seen when number of clusters is 2.

**Results:**

Distortions for k = 1 to 14 : [0.6723847012398, 0.520943856709, 0.4823057134, 0.4523059701247, 0.414097234191, 0.39092375093124, 0.381098723509, 0.369071324097178, 0.3478152340987, 0.3423094709255, 0.328913489619, 0.32203847209, 0.3180732508676, 0.30896132419029]




## Code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from scipy.spatial.distance import cdist
from sklearn.cluster import KMeans


df = pd.read_csv("Frogs_MFCCs.csv")

df.replace('Bufonidae',int(1),inplace=True)
df.replace('Dendrobatidae',int(2),inplace=True)
df.replace('Hylidae',int(3),inplace=True)
df.replace('Leptodactylidae',int(4),inplace=True)

df.replace('Adenomera',int(5),inplace=True)
df.replace('Ameerega',int(6),inplace=True)
df.replace('Dendropsophus',int(7),inplace=True)
df.replace('Hypsiboas',int(8),inplace=True)
df.replace('Leptodactylus',int(9),inplace=True)
df.replace('Osteocephalus',int(10),inplace=True)
df.replace('Rhinella',int(11),inplace=True)
df.replace('Scinax',int(12),inplace=True)

df.replace('AdenomeraAndre',int(13),inplace=True)
df.replace('AdenomeraHylaedactylus',int(14),inplace=True)
df.replace('Ameeregatrivittata',int(15),inplace=True)
df.replace('HylaMinuta',int(16),inplace=True)
df.replace('HypsiboasCinerascens',int(17),inplace=True)
df.replace('HypsiboasCordobae',int(18),inplace=True)
df.replace('LeptodactylusFuscus',int(19),inplace=True)
df.replace('OsteocephalusOophagus',int(20),inplace=True)
df.replace('Rhinellagranulosa',int(21),inplace=True)
df.replace('ScinaxRuber',int(22),inplace=True)

df.drop('RecordID', inplace=True, axis=1)

df.to_csv('Frogs_MFCCs_new.csv', index=False)

shuffled_data = shuffle(df.values, random_state=0)
predictors = shuffled_data[:, :22]
class_label = shuffled_data[:, 22:]
```

```
number_of_clusters = range(1, 15)
variances = []
for k in number_of_clusters:
    model = KMeans(n_clusters=k)
    model.fit(predictors)
    variance = sum(np.min(cdist(predictors, model.cluster_centers_, 'euclidean'),
axis=1)) / predictors.shape[0]
    variances.append(variance)

plt.plot(number_of_clusters, variances, 'bx-')
plt.xlabel('K')
plt.ylabel('Distortion')
plt.show()

diff = []
biggest_diff = 0
k = 0
for i in range(0, len(variances)-1):
    difference_in_variances = np.abs(variances[i + 1] - variances[i])
    if difference_in_variances > biggest_diff:
        biggest_diff = difference_in_variances
        k = i

best_value_of_k = k+2
```

**b. K-mean Clustering:**

I have clustered the data into two clusters using k-mean clustering because the optimal value of *k* and determined the species and genus of each of these clusters by majority polling after determining the family of each cluster.

**Results:**

First Cluster:

Family = Hylidae
Genus = Hypsiboas
Species = HypsiboasCordobae

Second Cluster:

Family = Leptodactylidae
Genus = Adenomera
Species = AdenomeraHylaedactylus

As it can be seen in the results that the family of first cluster is Hylindae, genus is Hypsiboas and species is HypsiboasCordobae, whereas the family of second cluster is Leptodactylidae, genus is Adenomera and species is AdenomeraHylaedactylus.

## Code:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.cluster import KMeans
import random


df = pd.read_csv("Frogs_MFCCs.csv")

df.replace('Bufonidae',int(1),inplace=True)
df.replace('Dendrobatidae',int(2),inplace=True)
df.replace('Hylidae',int(3),inplace=True)
df.replace('Leptodactylidae',int(4),inplace=True)

df.replace('Adenomera',int(5),inplace=True)
df.replace('Ameerega',int(6),inplace=True)
df.replace('Dendropsophus',int(7),inplace=True)
df.replace('Hypsiboas',int(8),inplace=True)
df.replace('Leptodactylus',int(9),inplace=True)
df.replace('Osteocephalus',int(10),inplace=True)
df.replace('Rhinella',int(11),inplace=True)
df.replace('Scinax',int(12),inplace=True)

df.replace('AdenomeraAndre',int(13),inplace=True)
df.replace('AdenomeraHylaedactylus',int(14),inplace=True)
df.replace('Ameeregatrivittata',int(15),inplace=True)
df.replace('HylaMinuta',int(16),inplace=True)
df.replace('HypsiboasCinerascens',int(17),inplace=True)
df.replace('HypsiboasCordobae',int(18),inplace=True)
df.replace('LeptodactylusFuscus',int(19),inplace=True)
df.replace('OsteocephalusOophagus',int(20),inplace=True)
df.replace('Rhinellagranulosa',int(21),inplace=True)
df.replace('ScinaxRuber',int(22),inplace=True)

df.drop('RecordID',inplace=True,axis=1)
df.to_csv('Frogs_MFCCs_new.csv',index=False)
shuffled_data = shuffle(df.values, random_state=5)

training_x, testing_x, training_y, testing_y =
train_test_split(shuffled_data[:,:22], shuffled_data[:,22:], train_size=0.7)

family_dataset_Y = shuffled_data[:, 22:][:, :1]
genus_dataset_Y = shuffled_data[:, 22:][:, 1:2]
species_dataset_Y = shuffled_data[:, 22:][:, 2:]

model = KMeans(n_clusters=2, n_init=20, random_state=5)
model.fit(shuffled_data[:,:22])
labels = model.labels_
print(model.cluster_centers_)
print(model.inertia_)

family_count_for_first_cluster = {1:0, 2:0, 3:0, 4:0}
family_count_for_second_cluster = {1:0, 2:0, 3:0, 4:0}
for i in range(1, len(labels)):
    if int(labels[i]) == 0:
        family_count_for_first_cluster[int(family_dataset_Y[i])] =
family_count_for_first_cluster[int(family_dataset_Y[i])] + 1
    else:
```

```
        family_count_for_second_cluster[int(family_dataset_Y[i])] =
family_count_for_second_cluster[int(family_dataset_Y[i])] + 1

majority_family_for_1_index = int(max(family_count_for_first_cluster.keys(),
key=(lambda label: family_count_for_first_cluster[label])))
majority_family_for_2_index = int(max(family_count_for_second_cluster.keys(),
key=(lambda label: family_count_for_second_cluster[label])))

family_labels = {1: 'Bufonidae', 2:
'Dendrobatidae',3:'Hylidae',4:'Leptodactylidae'}
family_of_first_cluster = family_labels[majority_family_for_1_index]
family_of_second_cluster = family_labels[majority_family_for_2_index]


genus_count_for_first_cluster = {5:0, 6:0, 7:0, 8:0, 9:0, 10:0, 11:0, 12:0}
genus_count_for_second_cluster = {5:0, 6:0, 7:0, 8:0, 9:0, 10:0, 11:0, 12:0}
for i in range(1, len(labels)):
    if int(labels[i]) == 0:
        genus_count_for_first_cluster[int(genus_dataset_Y[i])] =
genus_count_for_first_cluster[int(genus_dataset_Y[i])] + 1
    else:
        genus_count_for_second_cluster[int(genus_dataset_Y[i])] =
genus_count_for_second_cluster[int(genus_dataset_Y[i])] + 1

majority_genus_for_1_index = int(max(genus_count_for_first_cluster.keys(),
key=(lambda label: genus_count_for_first_cluster[label])))
majority_genus_for_2_index = int(max(genus_count_for_second_cluster.keys(),
key=(lambda label: genus_count_for_second_cluster[label])))

genus_labels =
{5:'Adenomera',6:'Ameerega',7:'Dendropsophus',8:'Hypsiboas',9:'Leptodactylus',10:'O
steocephalus',11:'Rhinella',12:'Scinax'}
genus_of_first_cluster = genus_labels[majority_genus_for_1_index]
genus_of_second_cluster = genus_labels[majority_genus_for_2_index]


species_count_for_first_cluster = {13:0, 14:0, 15:0, 16:0, 17:0, 18:0, 19:0, 20:0,
21:0, 22:0}
species_count_for_second_cluster = {13:0, 14:0, 15:0, 16:0, 17:0, 18:0, 19:0, 20:0,
21:0, 22:0}

for i in range(1, len(labels)):
    if int(labels[i]) == 1:
        species_count_for_first_cluster[int(species_dataset_Y[i])] =
species_count_for_first_cluster[int(species_dataset_Y[i])] + 1
    else:
        species_count_for_second_cluster[int(species_dataset_Y[i])] =
species_count_for_second_cluster[int(species_dataset_Y[i])] + 1

majority_species_for_1_index = int(max(species_count_for_first_cluster.keys(),
key=(lambda label: species_count_for_first_cluster[label])))
majority_species_for_2_index = int(max(species_count_for_second_cluster.keys(),
key=(lambda label: species_count_for_second_cluster[label])))

species_labels =
{13:'AdenomeraAndre',14:'AdenomeraHylaedactylus',15:'Ameeregatrivittata',16:'HylaMi
nuta',17:'HypsiboasCinerascens',18:'HypsiboasCordobae',19:'LeptodactylusFuscus',20:
'OsteocephalusOophagus',21:'Rhinellagranulosa',22:'ScinaxRuber'}
species_of_first_cluster = species_labels[majority_species_for_1_index]
species_of_second_cluster = species_labels[majority_species_for_2_index]
```

## c. Hamming Loss of the Majority Triplets:

I have calculated the hamming loss by comparing the true labels to the labels predicted by clustering and then, calculated the mean of these hamming losses to come up with an average hamming loss of the given dataset.

**Results:**

Family of first cluster = Hylidae
Family of second cluster = Leptodactylidae
Hamming Loss = 0.2309237535

Genus of first cluster = Hypsiboas
Genus of second cluster = Adenomera
Hamming Loss = 0.29915674786

Species of first cluster = HypsiboasCordobae
Species of second cluster = AdenomeraHylaedactylus
Hamming Loss = 0.372407918324

Average Hamming Loss = 0.29233209406812

**Majority Triplet of first cluster:**
Family = Hylidae, Genus = Hypsiboas, Species = HypsiboasCordobae

**Majority Triplet of second cluster:**
Family = Leptodactylidae, Genus = Adenomera, Species = AdenomeraHylaedactylus

## Code:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.cluster import KMeans
from sklearn.metrics import hamming_loss


df = pd.read_csv("Frogs_MFCCs.csv")

df.replace('Bufonidae',int(1),inplace=True)
df.replace('Dendrobatidae',int(2),inplace=True)
df.replace('Hylidae',int(3),inplace=True)
df.replace('Leptodactylidae',int(4),inplace=True)

df.replace('Adenomera',int(5),inplace=True)
df.replace('Ameerega',int(6),inplace=True)
df.replace('Dendropsophus',int(7),inplace=True)
df.replace('Hypsiboas',int(8),inplace=True)
df.replace('Leptodactylus',int(9),inplace=True)
df.replace('Osteocephalus',int(10),inplace=True)
df.replace('Rhinella',int(11),inplace=True)
df.replace('Scinax',int(12),inplace=True)

df.replace('AdenomeraAndre',int(13),inplace=True)
df.replace('AdenomeraHylaedactylus',int(14),inplace=True)
df.replace('Ameeregatrivittata',int(15),inplace=True)
df.replace('HylaMinuta',int(16),inplace=True)
df.replace('HypsiboasCinerascens',int(17),inplace=True)
```

```
df.replace('HypsiboasCordobae',int(18),inplace=True)
df.replace('LeptodactylusFuscus',int(19),inplace=True)
df.replace('OsteocephalusOophagus',int(20),inplace=True)
df.replace('Rhinellagranulosa',int(21),inplace=True)
df.replace('ScinaxRuber',int(22),inplace=True)

df.drop('RecordID',inplace=True,axis=1)
df.to_csv('Frogs_MFCCs_new.csv',index=False)
shuffled_data = shuffle(df.values, random_state=5)

training_x, testing_x, training_y, testing_y = train_test_split(shuffled_data[:,:22],
shuffled_data[:,22:], train_size=0.7)

family_dataset_Y = shuffled_data[:, 22:][:, :1]
genus_dataset_Y = shuffled_data[:, 22:][:, 1:2]
species_dataset_Y = shuffled_data[:, 22:][:, 2:]

model = KMeans(n_clusters=2, n_init=20, random_state=5)
model.fit(shuffled_data[:,:22])
labels = model.labels_
print(model.cluster_centers_)
print(model.inertia_)

family_count_for_first_cluster = {1:0, 2:0, 3:0, 4:0}
family_count_for_second_cluster = {1:0, 2:0, 3:0, 4:0}
for i in range(1, len(labels)):
    if int(labels[i]) == 0:
        family_count_for_first_cluster[int(family_dataset_Y[i])] =
family_count_for_first_cluster[int(family_dataset_Y[i])] + 1
    else:
        family_count_for_second_cluster[int(family_dataset_Y[i])] =
family_count_for_second_cluster[int(family_dataset_Y[i])] + 1

majority_family_for_1_index = int(max(family_count_for_first_cluster.keys(),
key=(lambda label: family_count_for_first_cluster[label])))
majority_family_for_2_index = int(max(family_count_for_second_cluster.keys(),
key=(lambda label: family_count_for_second_cluster[label])))

family_labels = {1: 'Bufonidae', 2: 'Dendrobatidae',3:'Hylidae',4:'Leptodactylidae'}
family_of_first_cluster = family_labels[majority_family_for_1_index]
family_of_second_cluster = family_labels[majority_family_for_2_index]

family_predictions = np.arange(0, len(labels), 1)
for i in range(1, len(labels)):
    if labels[i] == 1:
        family_predictions[i] = majority_family_for_1_index
    else:
        family_predictions[i] = majority_family_for_2_index

family_hamming_loss = hamming_loss(family_dataset_Y,
family_predictions.reshape(len(family_predictions), 1))


genus_count_for_first_cluster = {5:0, 6:0, 7:0, 8:0, 9:0, 10:0, 11:0, 12:0}
genus_count_for_second_cluster = {5:0, 6:0, 7:0, 8:0, 9:0, 10:0, 11:0, 12:0}
for i in range(1, len(labels)):
    if int(labels[i]) == 0:
        genus_count_for_first_cluster[int(genus_dataset_Y[i])] =
genus_count_for_first_cluster[int(genus_dataset_Y[i])] + 1
    else:
        genus_count_for_second_cluster[int(genus_dataset_Y[i])] =
genus_count_for_second_cluster[int(genus_dataset_Y[i])] + 1

majority_genus_for_1_index = int(max(genus_count_for_first_cluster.keys(),
key=(lambda label: genus_count_for_first_cluster[label])))
majority_genus_for_2_index = int(max(genus_count_for_second_cluster.keys(),
```

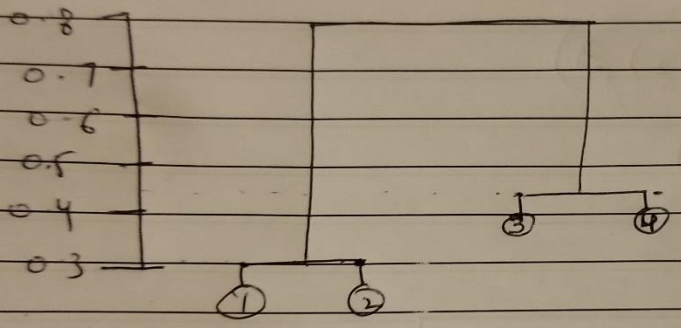
```
key=(lambda label: genus_count_for_second_cluster[label])))

genus_labels =
{5:'Adenomera',6:'Ameerega',7:'Dendropsophus',8:'Hypsiboas',9:'Leptodactylus',10:'Ost
eocephalus',11:'Rhinella',12:'Scinax'}
genus_of_first_cluster = genus_labels[majority_genus_for_1_index]
genus_of_second_cluster = genus_labels[majority_genus_for_2_index]

genus_predictions = np.arange(0, len(labels), 1)
for i in range(1, len(labels)):
    if labels[i] == 1:
        genus_predictions[i] = majority_genus_for_1_index
    else:
        genus_predictions[i] = majority_genus_for_2_index
genus_hamming_loss = hamming_loss(genus_dataset_Y,
genus_predictions.reshape(len(genus_predictions), 1))


species_count_for_first_cluster = {13:0, 14:0, 15:0, 16:0, 17:0, 18:0, 19:0, 20:0,
21:0, 22:0}
species_count_for_second_cluster = {13:0, 14:0, 15:0, 16:0, 17:0, 18:0, 19:0, 20:0,
21:0, 22:0}

for i in range(1, len(labels)):
    if int(labels[i]) == 1:
        species_count_for_first_cluster[int(species_dataset_Y[i])] =
species_count_for_first_cluster[int(species_dataset_Y[i])] + 1
    else:
        species_count_for_second_cluster[int(species_dataset_Y[i])] =
species_count_for_second_cluster[int(species_dataset_Y[i])] + 1

majority_species_for_1_index = int(max(species_count_for_first_cluster.keys(),
key=(lambda label: species_count_for_first_cluster[label])))
majority_species_for_2_index = int(max(species_count_for_second_cluster.keys(),
key=(lambda label: species_count_for_second_cluster[label])))

species_labels =
{13:'AdenomeraAndre',14:'AdenomeraHylaedactylus',15:'Ameeregatrivittata',16:'HylaMinu
ta',17:'HypsiboasCinerascens',18:'HypsiboasCordobae',19:'LeptodactylusFuscus',20:'Ost
eocephalusOophagus',21:'Rhinellagranulosa',22:'ScinaxRuber'}
species_of_first_cluster = species_labels[majority_species_for_1_index]
species_of_second_cluster = species_labels[majority_species_for_2_index]


species_predictions = np.arange(0, len(labels), 1)
for i in range(1, len(labels)):
    if labels[i] == 1:
        species_predictions[i] = majority_species_for_1_index
    else:
        species_predictions[i] = majority_species_for_2_index
species_hamming_loss = hamming_loss(species_dataset_Y,
species_predictions.reshape(len(species_predictions), 1))

average_hamming_loss = np.mean(np.array([family_hamming_loss, genus_hamming_loss,
species_hamming_loss]))
```

**Question 3:**

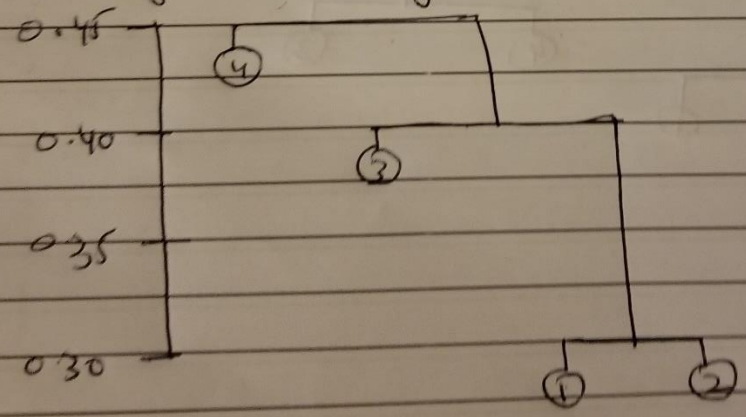
ISLR: 10.7.2:

a) Cluster Dendogram:

a) Complete linkage:




b) Single linkage:



c)

Cluster 1: (1,2)

Cluster 2: (3,4)

d)

Cluster 1: ((1,2),3)

Cluster 2: (4)

e) Complete: