# Data Models Problem Set III

Jacob Miller, Jacob Shiohira, Reid Gahan

Spring 2018, RAIK370H

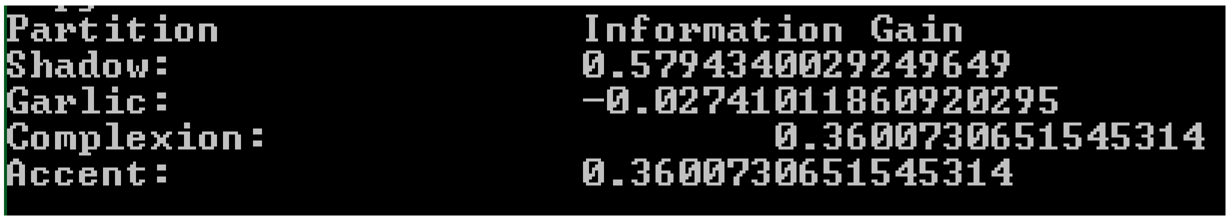
All code for this problem set is in Python. The approach for this problem set was to try and divide up the code by creating a function per question. The code for each problem is included directly following the problem number.

**Problem 1: Entropy Equation** Here is the code used for our entropy implementation with the vampire dataset:

```
'''
Prints the information gain values for each partition (or column).
'''
def runEntropy(df):

    # Get Y and labels:
    labels = list(df.axes[1])
    y_label = labels[0]
    Y = df[y_label]
    del labels[0]

    #Get information gain and loss for each partition:
    data_str = "Partition\t\tInformation Gain"
    for i in labels:
        data_str += "\n{}:\t\t\t{}".format(i, infoGainLoss(df[i],Y))

    #Print data
    print(data_str)

'''
Computes the entropy of the Y column.
'''
def getSetEntropy(Y):
    partitionEntropy = 0
    p_1 = sum(Y)/len(Y)
    p_2 = 1-p_1
    if p_2 == 0:
        partitionEntropy = 0
    elif p_1 == 0:
        partitionEntropy = 1
    else:
        partitionEntropy = -((p_1*math.log(p_1,2)) + (p_2*math.log(p_2,2)))
    return(partitionEntropy)

'''
Computes the information gain, which is simply the entropy of the set minus
the combined entropies of each result in a partition (or a column).
'''
def infoGainLoss(X,Y):
    setEntropy = getSetEntropy(Y)
```

```python
    part_dict = getPartitionEntropy(X,Y)
    set_xresults = set(X)
    part_fracs = set()
    for p in set_xresults:
        total = 0
        numPart = 0
        for i in range(len(X)):
            total += 1
            if X[i] == p:
                numPart += 1
        try:
            if not math.isnan(float(p)):
                part_fracs.add((numPart/total)*part_dict[p])
        except:
            part_fracs.add((numPart/total)*part_dict[p])
    return(setEntropy - sum(part_fracs))

'''
Gets Entropy for each value in a partitioned column and returns
them in a dictionary used to compute the combined partition entropy
in infoGainLoss.
'''
def getPartitionEntropy(X,Y):
    set_xresults = set(X)
    partitionEntropy = 0
    part_dict = {}
    for p in set_xresults:
        numTotal = 0
        numCorrect = 0
        for i in range(len(X)):
            if X[i] == p:
                numTotal += 1
                if Y[i] == 1:
                    numCorrect += 1
        if numTotal == 0:
            p_1 = 0
            p_2 = 1
        else:
            p_1 = numCorrect/numTotal
            p_2 = 1-p_1

        if p_2 == 0:
            partitionEntropy = 0
        elif p_1 == 0:
            partitionEntropy = 1
        else:
            partitionEntropy = -((p_1*math.log(p_1,2)) + (p_2*math.log(p_2,2)))

        part_dict[p] = partitionEntropy
    return part_dict
```

Here is the information gain for each partition in the Vampire data set:



```
Partition                   Information Gain
Shadow:                     0.5794340029249649
Garlic:                     -0.02741011860920295
Complexion:                         0.3600730651545314
Accent:                     0.3600730651545314
```

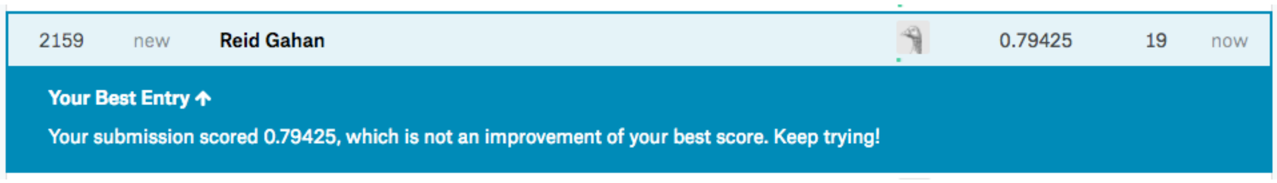
**Problem 2: Kaggle Titanic**

Here is the code used for our Titanic Dataset implementation:

```python
def titanic():
    # Get training data
    filename = "../data/titanic_train.csv"
    train = load_data_frame(filename)

    # Split to X and Y
    X_train, Y_train = split_data_frame(train, "Survived")

    X_train_imp = prepare_titanic(X_train)

    # Create random Forest Classifier, fit
    clf = RandomForestClassifier(n_estimators=100)
    clf.fit(X_train_imp, Y_train)

    # Get testing data
    filename = "../data/titanic_test.csv"
    X_test = load_data_frame(filename)

    X_test_imp = prepare_titanic(X_test)

    X_test_imp["Deck_T"] = 0
    X_test_imp["Title_Royalty"] = 0

    # Store test predictions as Y
    Y_test = clf.predict(X_test_imp)

    # Convert Pandas Series to Numpy Array
    passenger_id = np.array(X_test["PassengerId"], dtype=pd.Series)

    # Combine Passenger_id with Y_test
    answer = np.column_stack((passenger_id, Y_test))

    # Save output to csv file
    np.savetxt("../data/titanic_output.csv",
               answer,
               fmt="%.0f",
               delimiter=",",
               header="PassengerId,Survived")

def split_data_frame(df, label):
    X = df.drop([label], axis=1)
    Y = df[label]
    return X, Y
```

```
def prepare_titanic(df):
    # Encode the Sex and Embarked objects
    df = dummy_column(df, "Sex")
    df = dummy_column(df, "Embarked")
    df = dummy_column(df, "Pclass")
    df["Fare"] = df["Fare"].fillna(df.Fare.mean())

    # Convert objects to useable encoded value
    df = convert_title(df)
    df = dummy_column(df, "Title")

    _group = df.groupby(["Sex", "Pclass", "Title"])
    group_median = _group.median()

    df["Age"] = df.apply(
            lambda x: fill_ages(x, group_median) if np.isnan(x["Age"]) else x["Age"], axis=1)
    df = convert_deck(df)
    df = process_family(df)

    # Drop the objects that can't be encoded, convert floats to float32
    df = df.drop(["Ticket", "Embarked", "Sex", "Pclass", "Cabin", "PassengerId", "Name",
              "Deck", "Title", "FamilySize"], axis=1).astype(np.float32)

    return df

def dummy_column(df, label):
    dummies = pd.get_dummies(df[label], prefix=label)
    df = pd.concat([df, dummies], axis=1)

    return df

def convert_title(df):
    df["Title"] = df["Name"].map(lambda x: x.split(",")[1].split(".")[0].strip())

    title_list = {
        "Mrs": "Mrs",
        "Mr": "Mr",
        "Master": "Master",
        "Miss": "Miss",
        "Major": "Officer",
        "Rev": "Officer",
        "Dr": "Officer",
        "Ms": "Mrs",
        "Mlle": "Miss",
        "Col": "Officer",
        "Capt": "Officer",
        "Mme": "Mrs",
        "Countess": "Royalty",
        "Don": "Royalty",
        "Jonkheer": "Royalty"
    }

    df["Title"] = df["Title"].map(title_list)
    return df


def fill_ages(row, grouped_median):
    if row['Sex'] == 'female' and row['Pclass'] == 1:
        if row['Title'] == 'Miss':
            return grouped_median.loc['female', 1, 'Miss']['Age']
        elif row['Title'] == 'Mrs':
            return grouped_median.loc['female', 1, 'Mrs']['Age']
        elif row['Title'] == 'Officer':
            return grouped_median.loc['female', 1, 'Officer']['Age']
        elif row['Title'] == 'Royalty':
            return grouped_median.loc['female', 1, 'Royalty']['Age']

    elif row['Sex'] == 'female' and row['Pclass'] == 2:
        if row['Title'] == 'Miss':
            return grouped_median.loc['female', 2, 'Miss']['Age']
        elif row['Title'] == 'Mrs':
            return grouped_median.loc['female', 2, 'Mrs']['Age']

    elif row['Sex'] == 'female' and row['Pclass'] == 3:
        if row['Title'] == 'Miss':
            return grouped_median.loc['female', 3, 'Miss']['Age']
        elif row['Title'] == 'Mrs':
            return grouped_median.loc['female', 3, 'Mrs']['Age']

    elif row['Sex'] == 'male' and row['Pclass'] == 1:
        if row['Title'] == 'Master':
            return grouped_median.loc['male', 1, 'Master']['Age']
        elif row['Title'] == 'Mr':
            return grouped_median.loc['male', 1, 'Mr']['Age']
        elif row['Title'] == 'Officer':
            return grouped_median.loc['male', 1, 'Officer']['Age']
        elif row['Title'] == 'Royalty':
            return grouped_median.loc['male', 1, 'Royalty']['Age']

    elif row['Sex'] == 'male' and row['Pclass'] == 2:
        if row['Title'] == 'Master':
            return grouped_median.loc['male', 2, 'Master']['Age']
        elif row['Title'] == 'Mr':
            return grouped_median.loc['male', 2, 'Mr']['Age']
        elif row['Title'] == 'Officer':
            return grouped_median.loc['male', 2, 'Officer']['Age']

    elif row['Sex'] == 'male' and row['Pclass'] == 3:
        if row['Title'] == 'Master':
            return grouped_median.loc['male', 3, 'Master']['Age']
        elif row['Title'] == 'Mr':
            return grouped_median.loc['male', 3, 'Mr']['Age']


def convert_deck(df):
    # Fill NaN values with U for unkown
    df["Cabin"] = df["Cabin"].fillna("U")
    df["Deck"] = df["Cabin"].astype(str).str[0]

    df = dummy_column(df, "Deck")

    return df


def process_family(df):
    df["FamilySize"] = df["Parch"] + df["SibSp"]

    df["Alone"] = df.FamilySize.map(lambda x: 1 if x == 0 else 0)
    df["Small"] = df.FamilySize.map(lambda x: 1 if 0 < x < 4 else 0)
    df["Large"] = df.FamilySize.map(lambda x: 1 if 4 < x else 0)

    return df


def load_data_frame(filename):
    """1. Import data set"""
    dataframe = None
    try:
        dataframe = pd.read_csv(filename)
    except:
        print("No file found for " + filename + ". Exiting now.")
        sys.exit()
    return dataframe
```

The preceding code produced an accuracy of .79425 and a ranking of 2159.


2159 new Reid Gahan 0.79425 19 now

**Your Best Entry**

Your submission scored 0.79425, which is not an improvement of your best score. Keep trying!

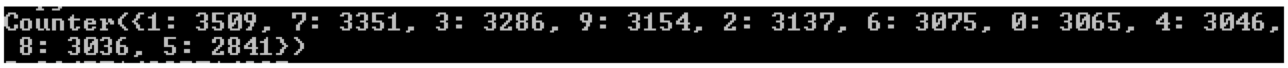
## Problem 3: Kaggle MNIST

Here is the code for our MNIST Digit Recognizer implementation:

```
def runRandomForest():
    # read train data
    train = pd.read_csv("../data/mnist/train.csv")
    train_df, test_df = train_test_split(train, test_size = 0.25)

    # Print labels to ensure equal numbers
    print(Counter(list(train_df['label'])))

    # read test data
    test = pd.read_csv("../data/mnist/test.csv")

    # Use only split training set for problem set
```

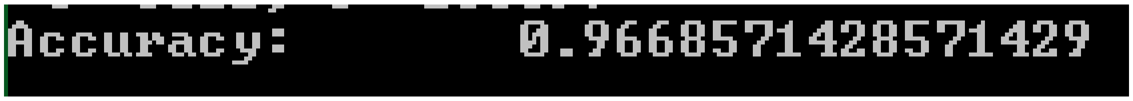
```
rand_for1 = RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state = 1)
rand_for1.fit(train_df.drop('label', axis=1),train_df['label'])
predictions1 = rand_for1.predict(test_df.drop('label', axis=1))

# Find accuracy
test_results = list(test_df['label'])
correct = 0
for i in range(len(predictions1)):
    if predictions1[i] == test_results[i]:
        correct += 1
accuracy = correct/len(test_df)
data_str = "Accuracy:\t{}".format(accuracy)
print(data_str)

# Create confusion table
cm = confusion_matrix(y_true = list(test_df['label']), y_pred = predictions1, labels=[0,1,2,3,4,5,6
plt.figure()
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(len([0,1,2,3,4,5,6,7,8,9]))
plt.xticks(tick_marks, [0,1,2,3,4,5,6,7,8,9], rotation=45)
plt.yticks(tick_marks, [0,1,2,3,4,5,6,7,8,9])

fmt = '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()

# Use all training set data for Kaggle challenge
rand_for2 = RandomForestClassifier(n_estimators=100, n_jobs=2, random_state = 1)
rand_for2.fit(train.drop('label', axis=1),train['label'])
predictions2 = rand_for2.predict(test)

numId = []
for i in range(len(test)):
    numId.append(i+1)
answer = np.column_stack([numId, predictions2])

np.savetxt("../data/digit_recognizer_output_2.csv", answer, fmt="%.0f", delimiter=",", header="Imag
```

Counter for training set split to make sure all values are distributed about equally:



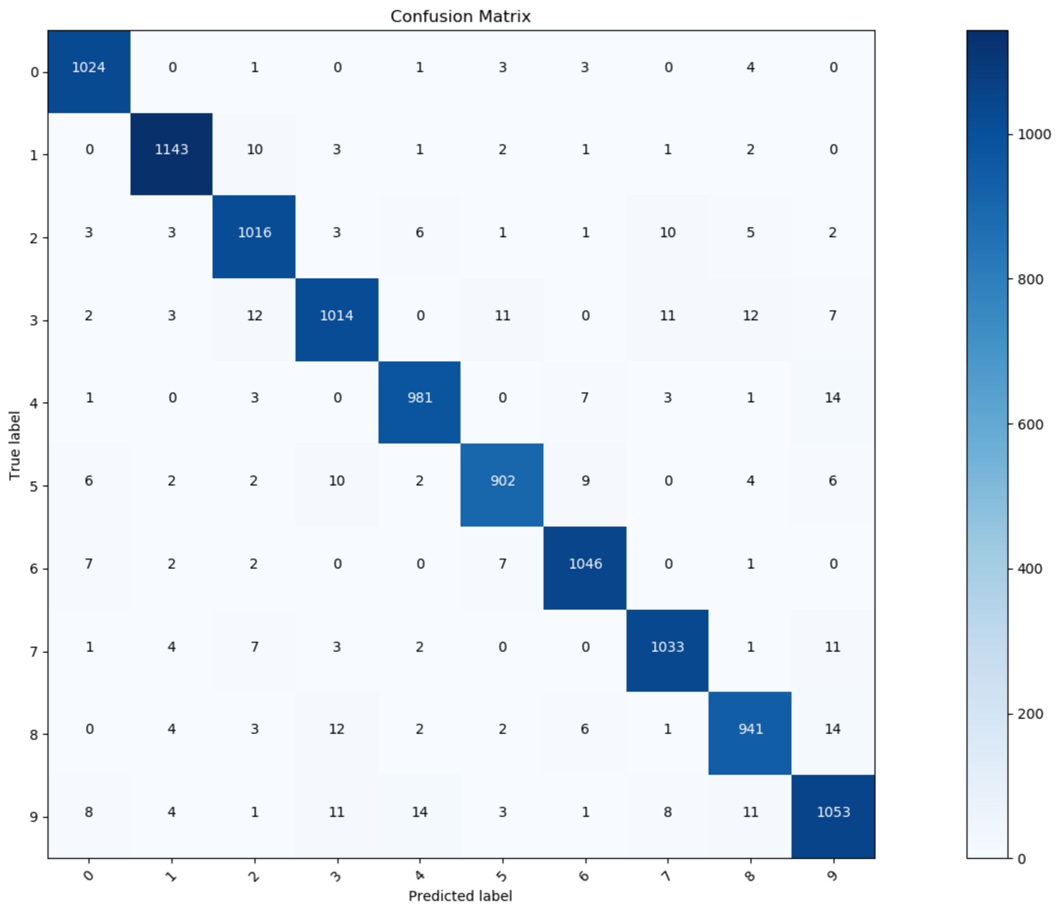
```
Counter({1: 3509, 7: 3351, 3: 3286, 9: 3154, 2: 3137, 6: 3075, 0: 3065, 4: 3046,
 8: 3036, 5: 2841})
```

Accuracy while splitting the training set data to 75% to train and 25% to test:



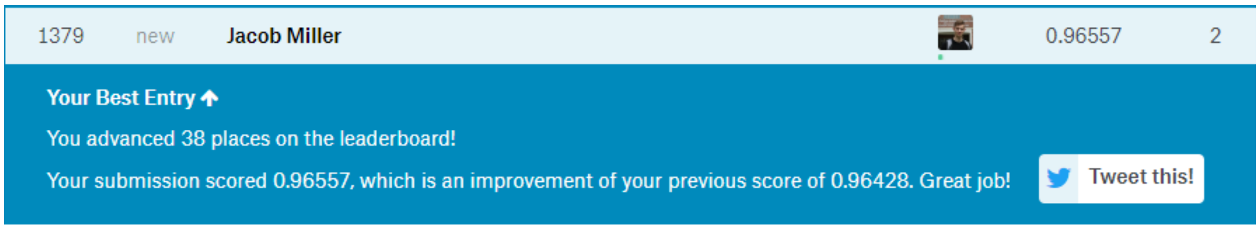
```
Accuracy:        0.9668571428571429
```

Confusion Matrix while splitting the training set data to 75% to train and 25% to test:



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1024 | 0 | 1 | 0 | 1 | 3 | 3 | 0 | 4 | 0 |
| 1 | 0 | 1143 | 10 | 3 | 1 | 2 | 1 | 1 | 2 | 0 |
| 2 | 3 | 3 | 1016 | 3 | 6 | 1 | 1 | 10 | 5 | 2 |
| 3 | 2 | 3 | 12 | 1014 | 0 | 11 | 0 | 11 | 12 | 7 |
| 4 | 1 | 0 | 3 | 0 | 981 | 0 | 7 | 3 | 1 | 14 |
| 5 | 6 | 2 | 2 | 10 | 2 | 902 | 9 | 0 | 4 | 6 |
| 6 | 7 | 2 | 2 | 0 | 0 | 7 | 1046 | 0 | 1 | 0 |
| 7 | 1 | 4 | 7 | 3 | 2 | 0 | 0 | 1033 | 1 | 11 |
| 8 | 0 | 4 | 3 | 12 | 2 | 2 | 6 | 1 | 941 | 14 |
| 9 | 8 | 4 | 1 | 11 | 14 | 3 | 1 | 8 | 11 | 1053 |

After training the model with all of the training data, and testing on the Kaggle test file, our resulting accuracy was .96557 with a rank of 1379:



1379 new **Jacob Miller** 0.96557 2

**Your Best Entry** ↑

You advanced 38 places on the leaderboard!

Your submission scored 0.96557, which is an improvement of your previous score of 0.96428. Great job!

There are three predicted numbers that tied for the 'most confused' number. A predicted 9 was confused for 8 fourteen times, again, a predicted 9 was confused with 4 fourteen times, and a predicted 4 was confused for a 9 fourteen times.

Insights:

One insight I gained while implementing the random forest was that there is a lot of good documentation for the python implementation. You can even optimize the performance of the model by setting the number of jobs for each processing core, and by setting the number of trees per forest. This is how I was able to run the whole training data through the model. If you increase the number of trees too much, then it will slow performance because each tree has to be evaluated. The improvement past 100 trees was very slim.

## Problem 4: K-Means

Here is the code for our custom K-Means implementation:

```
def runKMeans(filename):
    k = 3
    epsilon = 0.01
    max_iterations = 10
    seed = 10
    normalize = True
    dataframe, classes, original_dataframe = parse_arff_file(filename, normalize)

    rows, columns = dataframe.shape
    clusters, original_centroids,
    final_centroids, num_iterations,
    runtime, error = kmeans.k_means_clustering(dataframe, k, max_iterations,
                          epsilon, seed)

    print_k_means_data(original_dataframe, clusters, original_centroids,
                       final_centroids, num_iterations, runtime, error, classes)
    plot_results(clusters)

def k_means_clustering(dataframe, k, max_iterations, epsilon, seed):
    num_iterations = 1
    rows, columns = dataframe.shape
    old_sse, new_sse = sys.maxsize, 0
    df_as_arr = dataframe.reset_index().values
    min_max = [(dataframe[column].min(), dataframe[column].max()) for column in list(dataframe)]

    np.random.seed(seed)
    centroids = {
        i: [np.random.uniform(min_max[idx][0], min_max[idx][1]) for idx in range(len(list(dataframe)))]
        for i in range(k)
    }

    original_centroids = copy.deepcopy(centroids)

    start = time.time()
    while(num_iterations <= max_iterations):
        clusters = {}

        for instance in df_as_arr:
            instance_idx = instance[0]
            instance = instance[1:]
            min_dist, cluster_id = dist(instance, centroids)
            if cluster_id not in clusters.keys():
                clusters[cluster_id] = list()
            clusters[cluster_id].append([instance_idx, instance])

        old_sse = new_sse
        new_sse = 0
        for k,list_of_instances in clusters.items():
            centroid_sum = np.zeros(columns)
            centroid_size = len(list_of_instances)

            for instance in list_of_instances:
                instance = instance[1:]
                centroid_sum = np.add(centroid_sum, instance)

            centroids[k] = [(centroid_sum[i] / centroid_size) for i in range(len(centroid_sum))]
            current_error = [np.linalg.norm((np.array(instance[1:])-centroids[k]))**2
                             for instance in list_of_instances]
            new_sse += np.sum(current_error)

        if(math.fabs(old_sse - new_sse) < epsilon):
            end = time.time()
            return clusters, original_centroids, centroids, num_iterations, (end-start), new_sse

        num_iterations += 1

    end = time.time()
    return clusters, original_centroids, centroids, num_iterations, (end-start), new_sse

def dist(instance, centroids):
    min_dist = sys.maxsize
    min_dist_key = 0

    for k,v in centroids.items():
        d = np.linalg.norm(np.array(instance).reshape(1,-1)-np.array(v).reshape(1,-1))

        if d < min_dist:
            min_dist = d
            min_dist_key = k

    return min_dist, min_dist_key


def parse_arff_file(filename, normalize):
    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
    classes = []
    with open(filename) as file:
        df = a2p.load(file)

        try:
            df.iloc[:,-1] = df.iloc[:,-1].apply(int)
            classes = None
        except:
            le = preprocessing.LabelEncoder()
            le.fit(df.iloc[:,-1])
            classes = list(le.classes_)
            df.iloc[:,-1] = le.transform(df.iloc[:,-1])

        df = df.select_dtypes(include=numerics)

        original_dataframe = copy.deepcopy(df)
        if normalize:
            headers = df.columns
            x = df.values
            scaler = preprocessing.Normalizer()
            scaled_df = scaler.fit_transform(df)
            df = pd.DataFrame(scaled_df)
            df.columns=headers

        return df, classes, original_dataframe
```

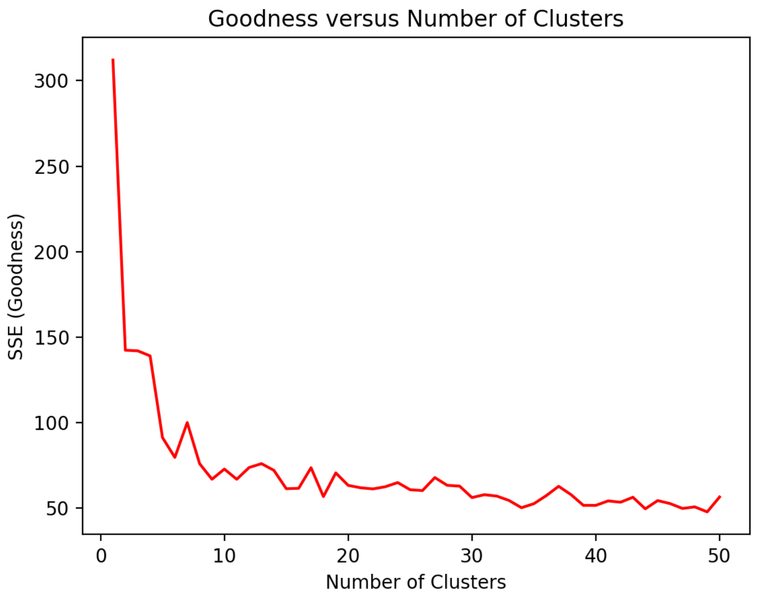
Here is the code for the built in SKLearn K-Means implementation:

```
from sklearn.cluster import KMeans
import generateKClusters

def builtInKMeans():
    filename = '../data/iris.arff'
    df, classes, original_dataframe = generateKClusters.parse_arff_file(filename, False)

    X = df.values
    start = time.time()
    kmeans = KMeans(n_clusters=3, random_state=0).fit(X)
    end = time.time()

    print("Time: {}".format(end - start))
```

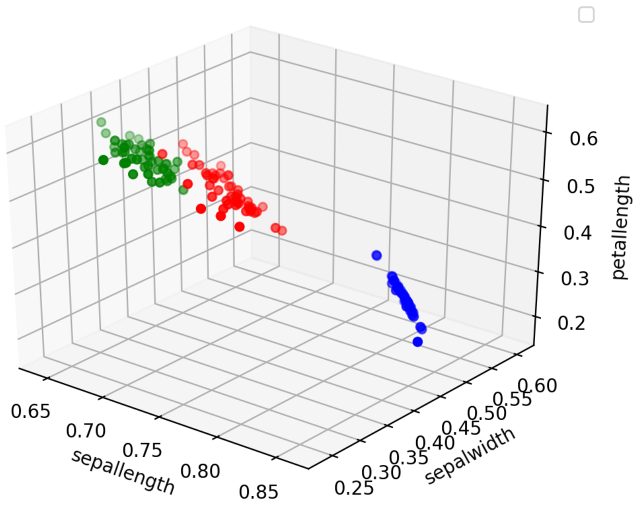
## Custom Implementation Approach

The actual k-means algorithm is not that complicated. There are 3 main steps that create initial centroids, calculate the closest centroid for all data points, update the centroids' coordinates based on the average value of all data points in a cluster, and calculate the SSE for the updated centroids. Some of the challenges of k-means include creating the initial centroids, efficiently computing distance, and keeping track of when to terminate the algorithm. Initial centroid creation is meant to be somewhat random because it is difficult to deduce optimal centroid locations. There has been research done on better techniques that have resulted in ideas such as kmeans++. This implementation uses a more random approach and takes a random number between the minimum and maximum for each attribute in the data set. A seed variable is also used when generating random numbers so that the random functions behave deterministically, and independent runs of the k-means algorithm can produce repeatable results. Distance calculations are typically very expensive, especially when calculating Euclidean Distance because of the squares and square roots. The cost of this calculation is further increased when the magnitude of the numbers being used is also increased. That is why it can be a good idea to normalize the data set. Working with numbers in the range of 0 to 1 is much better than working with numbers in an arbitrary range. This implementation actually does not use the normal Euclidean distance calculation because of the cost of it. Instead, the linalg package of the Python numpy library is used to calculate the norm of the distance between two vectors of instances of data. This resulted in an immense performance improvement.

Deciding the optimal number of clusters is difficult, but the elbow-method can be used to try and identify a “good” number of clusters. This method looks at the plot of sum of squared errors (SSE) versus number of clusters. The SSE is biased toward a larger number of clusters, so SSE is guaranteed to decrease. As the number of clusters approaches the number of instances in the data set, SSE approaches 0. At some point in the plot, there will be an inflection point from when the error rapidly decreases to decreases at a less rapid rate. This is a possible point for an optimal number of clusters.



**Cluster 3-D Scatter Plots**
Below is the visualization of with the three-axes of sepallength, sepalwidth, petallength. Each color represents a different cluster.



Below is the code to compare the runtime of our implementation versus the SKLearn built-in implementation.

```
def compareCustomAndBuiltIn(iterations=10):
    k = 3
    epsilon = 0.01
    max_iterations = 10
    seed = 10
    normalize = True
    filename = '../data/iris.arff'

    dataframe, classes, original_dataframe = generateKClusters.parse_arff_file(filename, normalize)

    X = dataframe.values
    total_custom_runtime = 0
    total_built_in_runtime = 0
    for i in range(iterations):
        clusters, original_centroids,
        final_centroids, num_iterations,
        runtime, error = kmeans.k_means_clustering(dataframe, k, max_iterations, epsilon, seed)
        total_custom_runtime += runtime

        start = time.time()
        KMeans(n_clusters=3, random_state=0).fit(X)
        end = time.time()
        total_built_in_runtime += (end-start)

    total_custom_runtime /= iterations
    total_built_in_runtime /= iterations
    print("""Custom average runtime for 10 iterations: {} seconds versus Built-in average runtime
           for 10 iterations: {} seconds""".format(round(total_custom_runtime,3),
                                               round(total_built_in_runtime,3)))
```

The output of the above function is the following: Custom average runtime for 10 iterations: 0.027 seconds versus Built-in average runtime for 10 iterations: 0.012 seconds.

**Problem 5: Naive Bayes with Mushrooms**

```
def naiveBayes():
    filename = '../data/mushroom/agaricus-lepiota.data'
    print_comparison = False
    df = load_data_frame(filename)
    rows, columns = df.shape
    print("There are {} rows and {} columns".format(rows, columns))
    describe_data_frame(df)

    df.loc[df['class'] == 'e'] = 0
    df.loc[df['class'] == 'p'] = 1

    Y = df['class']
    del df['class']
    X = df

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.25, random_state=10)

    print('Length of X training set: ' + str(len(X_train)))
    print('Length of Y training set: ' + str(len(Y_train)))
    print('Length of X testing set: ' + str(len(X_test)))
    print('Length of Y testing set: ' + str(len(Y_test)) + '\n')

    print("Predicting the training labels...")
    fitted_model = BernoulliNB().fit(X_train, Y_train)
    predictions = fitted_model.predict(X_train)
    print("Confusion Matrix: \n{}".format(confusion_matrix(Y_train, predictions)))
    print("\nAccuracy Score: {}".format(accuracy_score(Y_train, predictions)))

    print("\nPredicting the testing labels...")
    fitted_model = BernoulliNB().fit(X_train, Y_train)
    predictions = fitted_model.predict(X_test)
    print("Confusion Matrix: \n{}".format(confusion_matrix(Y_test, predictions)))
    print("\nAccuracy Score: {}".format(accuracy_score(Y_test, predictions)))
```

*What are the dimensions of the data set?* There are 8124 rows and 23 columns in the data set.

*What are the response and explanatory variables? What type are they?* There are 22 explanatory variables and are all chars representing an element of a class: cap-shape,cap-surface,cap-color,bruises?,odor,gill-attachment,gill-spacing,gill-size,gill-color,stalk-shape,stalk-root,stalk-surface-above-ring,stalk-surface-below-ring,stalk-color-above-ring,stalk-color-below-ring,veil-type,veil-color,ring-number,ring-type,spore-print-color,population,habitat. The response variable is the class saying whether or not the mushroom is edible or poisonous.

*How many mushrooms in the data set are edible and poisonous?* There are 4208 (51.8%) edible mushrooms and 3916 (48.2%) poisonous mushrooms.

*Should the explanatory variables be scaled?* No, the data set does not need to be scaled. Naive Bayes sets the priors based on the data you give it, so it will scale those priors to match the data.

*Split the data set into training and test sets with 75% of data in the training set. Print the dimensions of each set.* Length of X training set: 6093, Length of Y training set: 6093, Length of X testing set: 2031, Length of Y testing set: 2031.

*Print conditional probability table.* Below is the conditional probability table for the cap-shape attribute. The middle two columns list the $\mathbb{P}(\text{x} \mid \text{c})$ conditional probabilities such that $\mathbb{P}(\text{cap-shape.b} \mid \text{edible})$ is an example. Then, the bottom row is the $\mathbb{P}(\text{outcomes})$ such that $\mathbb{P}(\text{edible})$ is an example. Then, the last column shows the $\mathbb{P}(\text{outlook instance})$ conditional probabilities such that $\mathbb{P}(\text{cap-shape.b})$ is an example.

| | edible | poisonous | |
|---|---|---|---|
| cap-shape.b | 0.10 | 0.01 | 0.056 |
| cap-shape.c | 0.001 | 0.0 | 0.0005 |
| cap-shape.f | 0.38 | 0.37 | 0.39 |
| cap-shape.k | 0.054 | 0.14 | 0.10 |
| cap-shape.x | 0.46 | 0.41 | 0.45 |
| cap-shape.s | 0.01 | 0.76 | 0.004 |
| | 0.52 | 0.48 | |

The Naive Bayes model predicted the training labels with an accuracy score of 1.0 and produced the following confusion matrix:

| | |
|---|---|
| 3136 | 0 |
| 0 | 2957 |

The Naive Bayes model predicted the testing labels with an accuracy score of 1.0 and produced the following confusion matrix:

| | |
|---|---|
| 1072 | 0 |
| 0 | 959 |

*Why is this a good data set for Naive Bayes despite mushrooms not being especially interesting?* Naive Bayes is of course useful when we can assume that the features are conditionally independent. However, this assumption is hardly true. But, this data set is a good data set for Naive Bayes because the features are all nominal and discrete.