CS148 Project 2 - Binary Classification Comparative Methods

For this project we're going to attempt a binary classification of a dataset using multiple methods and compare results.

Our goals for this project will be to introduce you to several of the most common classification techniques, how to perform them and tweek parameters to optimize outcomes, how to produce and interpret results, and compare performance. You will be asked to analyze your findings and provide explanations for observed performance.

Specifically you will be asked to classify whether a **patient is suffering from heart disease** based on a host of potential medical factors.

DEFINITIONS

Binary Classification: In this case a complex dataset has an added 'target' label with one of two options. Your learning algorithm will try to assign one of these labels to the data.

Supervised Learning: This data is fully supervised, which means it's been fully labeled and we can trust the veracity of the labeling.

Background: The Dataset

For this exercise we will be using a subset of the UCI Heart Disease dataset, leveraging the fourteen most commonly used attributes. All identifying information about the patient has been scrubbed.

The dataset includes 14 columns. The information provided by each column is as follows:

- **age:** Age in years
- **sex:** (1 = male; 0 = female)
- **cp:** Chest pain type (0 = asymptomatic; 1 = atypical angina; 2 = non-anginal pain; 3 = typical angina)
- **trestbps:** Resting blood pressure (in mm Hg on admission to the hospital)
- **cholserum:** Cholestoral in mg/dl
- **fbs** Fasting blood sugar > 120 mg/dl (1 = true; 0 = false)
- **restecg:** Resting electrocardiographic results (0= showing probable or definite left ventricular hypertrophy by Estes' criteria; 1 = normal; 2 = having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV))
- **thalach:** Maximum heart rate achieved
- **exang:** Exercise induced angina (1 = yes; 0 = no)
- **oldpeakST:** Depression induced by exercise relative to rest
- **slope:** The slope of the peak exercise ST segment (0 = downsloping; 1 = flat; 2 = upsloping)
- **ca:** Number of major vessels (0-3) colored by flourosopy
- **thal:** 1 = normal; 2 = fixed defect; 7 = reversable defect
- **Sick:** Indicates the presence of Heart disease (True = Disease; False = No disease)

Loading Essentials and Helper Functions

```
[64] #Here are a set of libraries we imported to complete this assignment.
     #Feel free to use these or equivalent libraries for your implementation
     import numpy as np # linear algebra
     import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
     import matplotlib.pyplot as plt # this is used for the plot the graph
     import os
     import seaborn as sns # used for plot interactive graph.
     from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
     from sklearn import metrics
     from sklearn.svm import SVC
     from sklearn.linear_model import LogisticRegression
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.tree import DecisionTreeClassifier
     from sklearn.cluster import KMeans
     from sklearn.metrics import confusion_matrix
     import sklearn.metrics.cluster as smc
     from sklearn.model_selection import KFold


     from matplotlib import pyplot
     import itertools

     %matplotlib inline

     import random

     random.seed(42)
```

Part 1. Load the Data and Analyze

Let's first load our dataset so we'll be able to work with it. (correct the relative path if your notebook is in a different directory than the csv file.)

```
[65] heart = pd.read_csv("heartdisease.csv")
```

Now that our data is loaded, let's take a closer look at the dataset we're working with. Use the head method, the describe method, and the info method to display some of the rows so we can visualize the types of data fields we'll be working with.

```
[66] heart.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	sick
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	False
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	False
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	False
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	False
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	False

```
[67] heart.describe()
     # heart.info()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.366337	0.683168	0.966997	131.623762	246.264026	0.148515	0.528053	149.646865	0.326733	1.039604	1.399340	0.729373	2.313531
std	9.082101	0.466011	1.032052	17.538143	51.830751	0.356198	0.525860	22.905161	0.469794	1.161075	0.616226	1.022606	0.612277
min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.000000
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.000000
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.000000
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.000000

Sometimes data will be stored in different formats (e.g., string, date, boolean), but many learning methods work strictly on numeric inputs. Call the info method to determine the datafield type for each column. Are there any that are problemmatic and why?

The 'sick' feature is boolean (true/false) but we want it to be numerical for the sake of data science processing. The boolean values can be converted into 0/1 where 0 is false, and 1 is true.

Determine if we're dealing with any null values. If so, report on which columns?

```
[68] sample_incomplete_rows = heart[heart.isnull().any(axis=1)].head()
```

Thankfully, there aren't any null values here so there is no need for data imputation. I can see this by running the cell block directly above - there are no incomplete rows that show up.

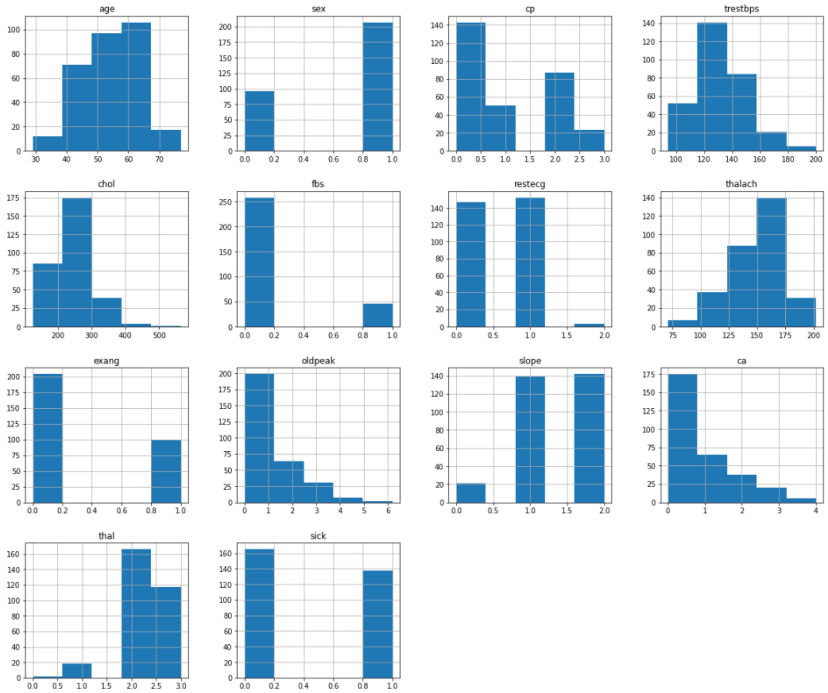
Before we begin our analysis we need to fix the field(s) that will be problematic. Specifically convert our boolean sick variable into a binary numeric target variable (values of either '0' or '1'), and then drop the original sick datafield from the dataframe. (hint: try label encoder or .astype()

```
[69] heart["sick"] = heart["sick"].astype(int)
     heart.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    int64
 1   sex       303 non-null    int64
 2   cp        303 non-null    int64
 3   trestbps  303 non-null    int64
 4   chol      303 non-null    int64
 5   fbs       303 non-null    int64
 6   restecg   303 non-null    int64
 7   thalach   303 non-null    int64
 8   exang     303 non-null    int64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    int64
 11  ca        303 non-null    int64
 12  thal      303 non-null    int64
 13  sick      303 non-null    int64
dtypes: float64(1), int64(13)
memory usage: 33.3 KB
```

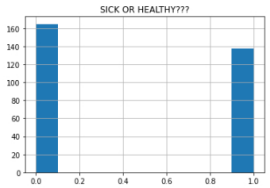
Now that we have a feel for the data-types for each of the variables, plot histograms of each field and attempt to ascertain how each variable performs (is it a binary, or limited selection, or does it follow a gradient?

```
[70] heart.hist(bins=5, figsize=(20, 17))
     plt.show()
```



We also want to make sure we are dealing with a balanced dataset. In this case, we want to confirm whether or not we have an equitable number of sick and healthy individuals to ensure that our classifier will have a sufficiently balanced dataset to adequately classify the two. Plot a histogram specifically of the sick target, and conduct a count of the number of sick and healthy individuals and report on the results:

```
[71] heart["sick"].hist()
     plt.title("SICK OR HEALTHY???")
     plt.show()

     heart["sick"].value_counts()
```




```
0    165
1    138
Name: sick, dtype: int64
```

Based on what we see above, we can conclude that there are 165 individuals who are not sick (they are healthy), and 138 individuals who are sick. This is sufficiently balanced as these two groups are pretty close in size.

Balanced datasets are important to ensure that classifiers train adequately and don't overfit, however arbitrary balancing of a dataset might introduce its own issues. Discuss some of the problems that might arise by artificially

balancing a dataset.

The main problem with artificially balancing a dataset is that it can introduce bias, intentionally or not. This bias could change the trends we observe in the data and lead to incorrect conclusions being drawn from the data. Datasets may be unbalanced because it indicates something relevant to the question we are trying to answer, so it is best not to balance datasets unless there is a very strong reason to do so.

Now that we have our dataframe prepared let's start analyzing our data. For this next question let's look at the correlations of our variables to our target value. First, map out the correlations between the values, and then discuss the relationships you observe. Do some research on the variables to understand why they may relate to the observed corellations. Intuitively, why do you think some variables correlate more highly than others (hint: one possible approach you can use the sns heatmap function to map the corr() method)?

```
[72] import matplotlib.pyplot as plt
     plt.figure(figsize=(16,12))
     sns.heatmap(heart.corr(method="pearson"), annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7efce4c884f0>
```

Negative Correlation

- There is a significant negative correlation between 'sick' and 'thalach' because if a patient has a low maximum heart rate it means their heart is less strong, and therefore less healthy. Thus they are prone to heart disease.
- There is a significant negative correlation between 'sick' and 'cp' because both typical and non-anigina pain are not related to the heart having issues.

Positive Correlation

- There is a significant positive correlation between 'sick' and 'ca' because when a person has multiple clogged arteries, it puts them at risk of heart disease. Arteries are very important to heart function so them being clogged is indicative of heart disease.
- There is a significant positive correlation between 'sick' and 'exang' because exercise induced angina is a symptom experienced by people with ischemic heart disease.

Of course, there are many more relationships between traits that can be observed but these are a few of the most notable relationships (in my opinion) that are related directly with heart disease.

Part 2. Prepare the 'Raw' Data and run a KNN Model

Before running our various learning methods, we need to do some additional prep to finalize our data. Specifically you'll have to cut the classification target from the data that will be used to classify, and then you'll have to divide the dataset into training and testing cohorts.

Specifically, we're going to ask you to prepare 2 batches of data: 1. Will simply be the raw numeric data that hasn't gone through any additional pre-processing. The other, will be data that you pipeline using your own selected methods. We will then feed both of these datasets into a classifier to showcase just how important this step can be!

Save the label column as a separate array and then drop it from the dataframe.

```
[3] heart_copy = heart["sick"].copy()
    heart_new = heart.drop("sick", axis=1)
```

First Create your 'Raw' unprocessed training data by dividing your dataframe into training and testing cohorts, with your training cohort consisting of 80% of your total dataframe (hint: use the train_test_split() method) Output the resulting shapes of your training and testing samples to confirm that your split was successful.

```
[4] from sklearn.model_selection import train_test_split
    x_train, x_test, y_train, y_test = train_test_split(heart_new, heart_copy, test_size=0.2, random_state=42)

    print("x_train shape:", x_train.shape)
    print("x_test shape:", x_test.shape)
    print("y_train shape:", y_train.shape)
    print("y_test shape:", y_test.shape)
```

```
x_train shape: (242, 13)
x_test shape: (61, 13)
y_train shape: (242,)
y_test shape: (61,)
```

We'll explore how not processing your data can impact model performance by using the K-Nearest Neighbor classifier. One thing to note was because KNN's rely on Euclidean distance, they are highly sensitive to the relative magnitude of different features. Let's see that in action! Implement a K-Nearest Neighbor algorithm on our raw data and report the results. For this initial implementation simply use the default settings. Refer to the KNN Documentation for details on implementation. Report on the accuracy of the resulting model.

```
[5] from sklearn.metrics import accuracy_score

    knn = KNeighborsClassifier().fit(x_train, y_train)
    knn_prediction = knn.predict(x_test)

    knn_accuracy = accuracy_score(y_test, knn_prediction)
```

```
[6] print("Accuracy:", knn_accuracy)

    # The accuracy here is not spectacular, but this result was using the default setttings so it should not be expected to be perfect. This is not a bad score considering how simple the above KNN algorithm is.
```

```
Accuracy: 0.6885245901639344
```

Now implement a pipeline of your choice. You can opt to handle categoricals however you wish, however please scale your numeric features using standard scaler. Use the fit_transform() to fit this pipeline to your training data. and then transform() to apply that pipeline to your test data

Pipeline:

```
[7] from sklearn.preprocessing import OneHotEncoder
    from sklearn.preprocessing import StandardScaler
    from sklearn.compose import ColumnTransformer

    x_train_drop = x_train.drop(["sex", "thal", "restecg", "cp"], axis=1)
    categories = ["sex", "thal", "restecg", "cp"]
    numbers = list(x_train_drop)

    pipeline = ColumnTransformer([("num", StandardScaler(), numbers), ("cat", OneHotEncoder(sparse=False), categories)])

    x_train_pipelined = pipeline.fit_transform(x_train)
    x_train_pipelined
```

```
array([[-1.35679832, -0.61685555,  0.91403366, ...,  1.        ,
         0.        ,  0.        ],
       [ 0.38508599,  1.1694912 ,  0.43952674, ...,  0.        ,
         0.        ,  0.        ],
       [-0.92132724,  1.1694912 , -0.30070405, ...,  0.        ,
         1.        ,  0.        ],
       ...,
       [ 1.58263146,  1.76494012, -0.24376322, ...,  0.        ,
         0.        ,  1.        ],
       [-0.92132724, -0.61685555,  0.04094093, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.92942484,  0.57404228, -0.98399402, ...,  1.        ,
         0.        ,  0.        ]])
```

```
[8] # Pipeline my test data
    x_test_pipelined = pipeline.transform(x_test)
```

Now retrain your model and compare the accuracy metrics with the raw and pipelined data.

```
[ ] # # k-Nearest Neighbors algorithm

    # from sklearn.metrics import accuracy_score

    # knn = KNeighborsClassifier().fit(x_train, y_train)
    # knn_prediction = knn.predict(x_test)

    # knn_accuracy = accuracy_score(y_test, knn_prediction)
```

```
[9] # x_train, x_test, y_train, y_test = train_test_split(x_train_pipelined, heart_copy, test_size=0.3, random_state=42)
    knnclf = KNeighborsClassifier()
    knnclf.fit(x_train_pipelined, y_train)
    y_predict = knnclf.predict(x_test_pipelined)
    metrics.accuracy_score(y_test, y_predict, normalize=True)
```

```
0.8688524590163934
```

The accuracy increased by roughly 18%. This means that transforming the data (removing nulls and scaling) properly provided cleaner data that the model was able to learn better from. This consequently improved the model's accuracy significantly.

Parameter Optimization. The KNN Algorithm includes an n_neighbors attribute that specifies how many neighbors to use when developing the cluster. (The default value is 5, which is what your previous model used.) Lets now try n values of: 1, 2, 3, 5, 7, 9, 10, 20, and 50. Run your model for each value and report the accuracy for each. (HINT leverage python's ability to loop to run through the array and generate results without needing to manually code each iteration).

```
[10] values = [1, 2, 3, 5, 7, 9, 10, 20, 50, 75, 100]
     for n in values:
         knnclf = KNeighborsClassifier(n_neighbors=n)
         knnclf.fit(x_train_pipelined, y_train)
         y_predict = knnclf.predict(x_test_pipelined)
         score = metrics.accuracy_score(y_test, y_predict, normalize=True)
         print('Accuracy for %d-Nearest-Neighbor: %.3f' %(n, score))
```

```
Accuracy for 1-Nearest-Neighbor: 0.820
Accuracy for 2-Nearest-Neighbor: 0.869
Accuracy for 3-Nearest-Neighbor: 0.885
Accuracy for 5-Nearest-Neighbor: 0.869
Accuracy for 7-Nearest-Neighbor: 0.836
Accuracy for 9-Nearest-Neighbor: 0.836
Accuracy for 10-Nearest-Neighbor: 0.852
Accuracy for 20-Nearest-Neighbor: 0.852
Accuracy for 50-Nearest-Neighbor: 0.869
Accuracy for 75-Nearest-Neighbor: 0.869
Accuracy for 100-Nearest-Neighbor: 0.852
```

Part 3. Additional Learning Methods

So we have a model that seems to work well. But let's see if we can do better! To do so we'll employ multiple learning methods and compare result.

Linear Decision Boundary Methods

Logistic Regression

Let's now try another classifier,one that's well known for handling linear models: Logistic Regression. Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable.

Implement a Logistical Regression Classifier. Review the Logistical Regression Documentation for how to implement the model.

This time in addition to accuracy report metrics for:

1. Accuracy
2. Precision
3. Recall
4. F1 Score

```
[73] # Logistic Regression

     from sklearn.metrics import classification_report
     from sklearn.linear_model import LogisticRegression

     logisticReg = LogisticRegression()
     logisticReg.fit(x_train_pipelined, y_train)
     y_predict = logisticReg.predict(x_test_pipelined)
     y_train_predict = logisticReg.predict(x_train_pipelined)

     print("Training Data \n")
     print(classification_report(y_train, y_train_predict))
```

```
print("Test Data \n")
print(classification_report(y_test, y_predict))
```

```
Training Data

              precision    recall  f1-score   support

       False       0.84      0.90      0.87       133
        True       0.87      0.79      0.83       109

    accuracy                           0.85       242
   macro avg       0.85      0.85      0.85       242
weighted avg       0.85      0.85      0.85       242

Test Data

              precision    recall  f1-score   support

       False       0.93      0.88      0.90        32
        True       0.87      0.93      0.90        29

    accuracy                           0.90        61
   macro avg       0.90      0.90      0.90        61
weighted avg       0.90      0.90      0.90        61
```

Discuss what each measure is reporting, why they are different, and why are each of these measures is significant.
Explore why we might choose to evaluate the performance of differing models differently based on these factors. Try to give some specific examples of scenarios in which you might value one of these measures over the others.

1. Accuracy represents the percentage of all labels that the model predicted correctly. This is generally a good measure to look at, as long as the dataset is reasonably balanced.
2. Precision refers to the number of true positives divided by the number of total positives. This is a good measure to look at when the consequences of a false positive are high.
3. Recall refers to the proportion of correctly predicted positive labels against all the labels in the class. This is an important measure to look at when false negatives have a higher consequence than a false postive (ie in cancer diagnosis).
4. F1 score is the weighted average of precision and recall. This measure gives a single score that measures the impact of both false positives and false negatives. It is especially useful to look at F1 score when data is unbalanced.

Let's tweak a few settings. First let's set your solver to 'sag', your max_iter= 10, and set penalty = 'none' and rerun your model. Let's see how your results change!

```
[74] # Logistic Regression
     from sklearn.metrics import classification_report
     from sklearn.linear_model import LogisticRegression

     logisticReg = LogisticRegression(solver='sag',max_iter=10)
     logisticReg.fit(x_train_pipelined, y_train)
     y_predict = logisticReg.predict(x_test_pipelined)
     y_train_predict = logisticReg.predict(x_train_pipelined)

     print("Training Data \n")
     print(classification_report(y_train, y_train_predict))

     print("Test Data \n")
     print(classification_report(y_test, y_predict))
```

```
Training Data

              precision    recall  f1-score   support

       False       0.84      0.92      0.88       133
        True       0.89      0.79      0.83       109

    accuracy                           0.86       242
   macro avg       0.86      0.85      0.86       242
weighted avg       0.86      0.86      0.86       242

Test Data

              precision    recall  f1-score   support

       False       0.93      0.88      0.90        32
        True       0.87      0.93      0.90        29

    accuracy                           0.90        61
   macro avg       0.90      0.90      0.90        61
weighted avg       0.90      0.90      0.90        61

/usr/local/lib/python3.8/dist-packages/sklearn/linear_model/_sag.py:352: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
```

Did you notice that when you ran the previous model you got the following warning: "ConvergenceWarning: The max_iter was reached which means the coef_ did not converge". Check the documentation and see if you can implement a fix for this problem, and again report your results.

```
[75] # Logistic Regression
     from sklearn.metrics import classification_report
     from sklearn.linear_model import LogisticRegression

     logisticReg = LogisticRegression(solver='sag',max_iter=1000)
     logisticReg.fit(x_train_pipelined, y_train)
     y_predict = logisticReg.predict(x_test_pipelined)
     y_train_predict = logisticReg.predict(x_train_pipelined)

     print("Training Data \n")
     print(classification_report(y_train, y_train_predict))

     print("Test Data \n")
     print(classification_report(y_test, y_predict))
```

```
Training Data

              precision    recall  f1-score   support

       False       0.84      0.90      0.87       133
        True       0.87      0.79      0.83       109

    accuracy                           0.85       242
   macro avg       0.85      0.85      0.85       242
weighted avg       0.85      0.85      0.85       242

Test Data

              precision    recall  f1-score   support

       False       0.93      0.88      0.90        32
        True       0.87      0.93      0.90        29

    accuracy                           0.90        61
   macro avg       0.90      0.90      0.90        61
weighted avg       0.90      0.90      0.90        61
```

Explain what you changed, and why do you think that may have altered the outcome.

I changed it from 'max_iter=10' to 'max_iter=1000'. The outcome is barely any different from when there was a convergence warning though. I believe that the model took many more iterations to finally converge, but the peak performance was reached shortly after the 10 iteration limit from before so the results don't really change.

Rerun your logistic classifier, but modify the penalty = 'l1', solver='liblinear' and again report the results.

```
[76] from sklearn.metrics import classification_report
     from sklearn.linear_model import LogisticRegression

     logisticReg = LogisticRegression(solver='liblinear',penalty='l1')
     logisticReg.fit(x_train_pipelined, y_train)
     y_predict = logisticReg.predict(x_test_pipelined)
     y_train_predict = logisticReg.predict(x_train_pipelined)

     print("Training Data \n")
     print(classification_report(y_train, y_train_predict))

     print("Test Data \n")
     print(classification_report(y_test, y_predict))
```

```
Training Data

              precision    recall  f1-score   support

       False       0.83      0.90      0.86       133
        True       0.87      0.77      0.82       109

    accuracy                           0.84       242
   macro avg       0.85      0.84      0.84       242
weighted avg       0.84      0.84      0.84       242

Test Data

              precision    recall  f1-score   support

       False       0.93      0.88      0.90        32
        True       0.87      0.93      0.90        29

    accuracy                           0.90        61
   macro avg       0.90      0.90      0.90        61
weighted avg       0.90      0.90      0.90        61
```

Explain what what the two solver approaches are, and why liblinear may have produced an improved outcome (but not always, and it's ok if your results show otherwise!).

SAG is an abbreviation for stochastic average gradient, which optimizes the sum of a finite number of smooth convex functions. SAG incorporates a memory of previous gradient values, which allows for faster convergence. This makes it a good option for faster convergence datasets.

LIBLINEAR uses automatic parameter selection and is mostly used on data sets with high dimensions. It is recommended for use in solving large-scale classification problems. This is more in line with the problem we are trying to solve here, which is why it may have produced a better outcome.

SVM (Support Vector Machine)

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimentional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

Implement a Support Vector Machine classifier on your pipelined data. Review the SVM Documentation for how to implement a model. For this implementation you can simply use the default settings, but set probability = True.

```
[54] # SVM
     from sklearn import svm

     svm_classifier = svm.SVC(probability=True)
     svm_classifier.fit(x_train_pipelined, y_train)
```

```
SVC(probability=True)
```

Report the accuracy, precision, recall, F1 Score, of your model, but in addition, plot a Confusion Matrix of your model's performance

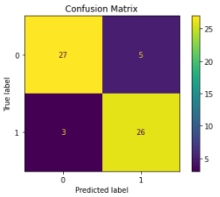
recommend using the `from sklearn.metrics import plot_confusion_matrix` library for this one!

```
[55] from sklearn import metrics
     from sklearn.svm import SVC

     svm_predict = svm_classifier.predict(x_test_pipelined)

     accuracy = metrics.accuracy_score(y_test, svm_predict)
     precision = metrics.precision_score(y_test, svm_predict)
     recall = metrics.recall_score(y_test, svm_predict)
     f1 = metrics.f1_score(y_test, svm_predict)


     print("Accuracy: {}".format(accuracy))
     print("Precision: {}".format(precision))
     print("Recall: {}".format(recall))
     print("F1: {}".format(f1))
```

```
Accuracy: 0.8688524590163934
Precision: 0.8387096774193549
Recall: 0.896551724137931
F1: 0.8666666666666666
```

```
[56] confusion = metrics.confusion_matrix(y_test, svm_predict)
     metrics.ConfusionMatrixDisplay(confusion).plot()
     plt.title('Confusion Matrix')
```

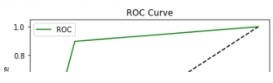
```
Text(0.5, 1.0, 'Confusion Matrix')
```

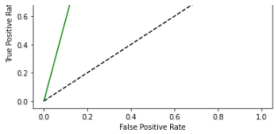


Plot a Receiver Operating Characteristic curve, or ROC curve, and describe what it is and what the results indicate

```
[57] # metrics.plot_roc_curve(svm_classifier, x_test, y_test)

     from sklearn.metrics import roc_curve

     false_positive_rate, true_positive_rate, threshold = roc_curve(y_test, svm_predict)
     print('AUC:', np.trapz(true_positive_rate, false_positive_rate))
     plt.plot([0, 1], [0, 1], color='black', linestyle='--')
     plt.plot(false_positive_rate, true_positive_rate, color='green', label='ROC')
     plt.legend()
     plt.title('ROC Curve')
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.show()
```

```
AUC: 0.8701508620689655
```





The ROC curve is a graph that we get by plotting the true positive rate against the false positive rate at different thresholds. The are under the ROC curve measures the usefulness of a given test. A greater area under the curve means that the test is more useful, so ROC curves can be easily compared to see which test is the most useful. The AUC is a value that indicates the model's performance. A higher value of AUC is desirable as it tends to label correctly.THe AUC here is higher than 0.5, which means that it is right some of the time but is still far from perfect (AUC = 1).

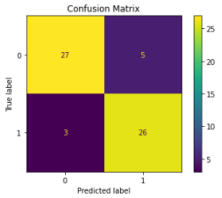
Rerun your SVM, but now modify your model parameter kernel to equal 'linear'. Again report your Accuracy, Precision, Recall, F1 scores, and Confusion matrix and plot the new ROC curve.

```
[61] # SVM
     svm_classifier_linear = svm.SVC(probability=True, kernel='linear')
     svm_classifier_linear.fit(x_train_pipelined, y_train)
```

```
SVC(kernel='linear', probability=True)
```

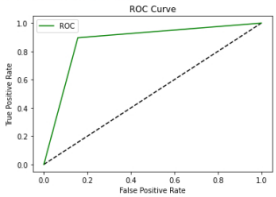
```
[62] svm_predict_linear = svm_classifier_linear.predict(x_test_pipelined)

     accuracy = metrics.accuracy_score(y_test, svm_predict_linear)
     precision = metrics.precision_score(y_test, svm_predict_linear)
     recall = metrics.recall_score(y_test, svm_predict_linear)
     f1 = metrics.f1_score(y_test, svm_predict_linear)


     print("Accuracy: {}".format(accuracy))
     print("Precision: {}".format(precision))
     print("Recall: {}".format(recall))
     print("F1: {}".format(f1))

     confusion = metrics.confusion_matrix(y_test, svm_predict_linear)
     metrics.ConfusionMatrixDisplay(confusion).plot()
     plt.title('Confusion Matrix')
```

```
Accuracy: 0.8688524590163934
Precision: 0.8387096774193549
Recall: 0.896551724137931
F1: 0.8666666666666666
Text(0.5, 1.0, 'Confusion Matrix')
```




```
[63] false_positive_rate, true_positive_rate, threshold = roc_curve(y_test, svm_predict_linear)
     print('AUC:', np.trapz(true_positive_rate, false_positive_rate))
     plt.plot([0, 1], [0, 1], color='black', linestyle='--')
     plt.plot(false_positive_rate, true_positive_rate, color='green', label='ROC')
     plt.legend()
     plt.title('ROC Curve')
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.show()
```

```
AUC: 0.8701508620689655
```



Explain the what the new results you've achieved mean. Read the documentation to understand what you've changed about your model and explain why changing that input parameter might impact the results in the manner you've observed.

My results seem like they haven't changed. I believe that they should have changed but I'm not sure why they didn't. Setting the kernel to 'linear' should have changed the way the model is trained, as it should have altered how it comes up with decision boundaries. By default the decision boundaries are determined radially (using polar coordinates). Canging it to the 'linear' setting should have changed the decision boundaries to a traditional coordinate system with straight lines. My guess is that linear was supposed to have poorer performance, but I'm not sure why my results appear identical in both cases.

Both logistic regression and linear SVM are trying to classify data points using a linear decision boundary, then what's the difference between their ways to find this boundary?

Logistic regression aims to minimize a cost function whereas SVM aims to find a separating hyperplane that maximizes the margin separating the different classes. They are two different approaches that are useful in different situations.