Part 1: Business Proposal

Daniel Yoon
dy38

What is the business objective of this project?

The business objective of this project is to help us predict a cancer diagnosis, specifically using breast cancer data. With a problem that affects us all so closely as humans, the benefit of this project is a direct improvement on our livelihoods. My model will aim to predict whether a cancer is malignant vs benign, and will it be possible to reduce the false negative rate? The data used for this project can be found here: https://www.kaggle.com/datasets/erdemtaha/cancer-data
With a reliable prediction, this model can improve diagnosing patients so they can make informed decisions on their health.

What assumptions are you making about the data and the problem?

The assumptions I'm making about this data is that the classification of malignant vs benign is accurate and not misdiagnosed. I also know the traits being observed are related to the cell nuclei of patients with breast cancer. The features that end with "_mean" are the mean of all the cells observed for that trait. Features that end with "_se" are the standard error, which is in respect to the mean. Features that end with "_worst" is the maximum value of those values observed. I also assume the units are in micrometers, since we are looking at human cells.
With respect to false diagnoses, it will be better to assume patients have cancer than not, since that will be less deadly in the long term; obviously reducing any false diagnoses overall will be a priority if possible. Striking a healthy balance between these will be necessary to improve my accuracy.

Is anything like this being done without Machine Learning? If so what?

Historically, yes there are things like this being done without Machine Learning but on a case by case basis. Because breast cancer has such a long history, many tools are used to help diagnose it like diffusion weighted imaging or spectroscopy. As seen in this link:
https://pmc.ncbi.nlm.nih.gov/articles/PMC8156889/
To study this in large amounts like how Machine Learning does, however, requires a tool with a large database. A product like Ancestry or 23andMe that gathers genetic information recognizes multiple factors that can determine how likely one is to have breast cancer without having to rely on machine learning.
https://www.23andme.com/

Part 2: Exploratory Data Analysis

Data Definition

field	type	description
id	int	unique identifier for each patient
diagnosis	str	Type of Cancer: "M" for Malignant or "B" for Benign
radius_mean	float	Mean of distances from the cells' nucleus to points on their perimeter. Measured in micrometers (μm).
texture_mean	float	Mean of the standard deviation of gray-scale values. Measured in micrometers (μm).
perimeter_mean	float	Mean of cells' perimeter. Measured in micrometers (μm).
area_mean	float	Mean of cells' area. Measured in micrometers (μm).
smoothness_mean	float	Mean of the local variation in the cells' radius lengths. Measured in micrometers (μm).
compactness_mean	float	Mean of the (perimeter^2 / area - 1.0)
concavity_mean	float	Mean of severity of the cells' concave portions of the contour. Measured in micrometers (μm).
concave points_mean	float	Mean of the number of cells' concave portions of the contour. Measured in micrometers (μm).
symmetry_mean	float	Mean of the cells' symmetry. Measured in micrometers (μm).
fractal_dimension_mean	float	Mean of the cells' (coastline approximation - 1). Measured in micrometers (μm).
radius_se	float	Standard error of distances from the cells' nucleus to points on their perimeter, in respect to the mean. Measured in micrometers (μm).
texture_se	float	Standard error of the standard deviation of gray-scale values, in respect to the mean. Measured in micrometers (μm).
perimeter_se	float	Standard error of cells' perimeter, in respect to the mean. Measured in micrometers (μm).
area_se	float	Standard error of cells' area, in respect to the mean. Measured in micrometers (μm).
smoothness_se	float	Standard error of the local variation in the cells' radius lengths, in respect to the mean. Measured in micrometers (μm).
compactness_se	float	Standard error of the (perimeter^2 / area - 1.0), in respect to the mean. Measured in micrometers (μm).
concavity_se	float	Standard error of severity of the cells' concave portions of the contour, in respect to the mean. Measured in micrometers (μm).
concave points_se	float	Standard error of the number of cells' concave portions of the contour, in respect to the mean. Measured in micrometers (μm).
symmetry_se	float	Standard error of the cells' symmetry, in respect to the mean. Measured in micrometers (μm).
fractal_dimension_se	float	Standard error of the cells' (coastline approximation - 1), in respect to the mean.

		Measured in micrometers (μm).
radius_worst	float	Worst of distances from the cells' nucleus to points on their perimeter. Measured in micrometers (μm).
texture_worst	float	Worst of the standard deviation of gray-scale values. Measured in micrometers (μm).
perimeter_worst	float	Worst of cells' perimeter. Measured in micrometers (μm).
area_worst	float	Worst of cells' area. Measured in micrometers (μm).
smoothness_worst	float	Worst of the local variation in the cells' radius lengths. Measured in micrometers (μm).
compactness_worst	float	Worst of the (perimeter^2 / area - 1.0)
concavity_worst	float	Worst of severity of the cells' concave portions of the contour. Measured in micrometers (μm).
concave points_worst	float	Worst of the number of cells' concave portions of the contour. Measured in micrometers (μm).
symmetry_worst	float	Worst of the cells' symmetry. Measured in micrometers (μm).
fractal_dimension_worst	float	Worst of the cells' (coastline approximation - 1). Measured in micrometers (μm).

Plan for missing data for each parameter?

None of the features have any missing values nor empty strings, so no plan is needed here. The features about concavity do have values of 0, but it is intentional as it indicates the cell has no concave characteristics.

Plan for additional parameters or data?

No extra parameters nor data is needed, as this dataset is robust on its own with 569 rows and 33 columns. Additionally, due to the extremely unique nature of this dataset, I could not find supplementary data that could be joined to this data set.

Any transformations needed?

Yes. My initial transformation will be to encode the diagnosis column, such that Malignant = 1 and Benign = 0. Since there are only two unique values here, this will be easy to encode without having to use the OrdinalEncoder library.

```
df = pd.read_csv("/Users/daniel/Desktop/Cancer_Data.csv")

print(f"Unique values in the 'diagnosis' column before encoding: {df['diagnosis'].unique()}")

df['diagnosis'] = df['diagnosis'].replace('B',0).replace('M', 1)

print(f"Unique values in the 'diagnosis' column after encoding: {df['diagnosis'].unique()}")
```

```
Unique values in the 'diagnosis' column before encoding: ['M' 'B']
Unique values in the 'diagnosis' column after encoding: [1 0]
```

In order to use the K nearest neighbors or K means clustering, I will need to normalize the features as they can have large differences. I will also need to omit these columns from the scaler, since they are irrelevant to the measurements of cells.

- id: Drop this column because it is a 1-1 mapping to each row. This 100% correlation will not contribute to training the model.

```
print(f"Total rows in the data set: {len(df)}")
print(f"Total count of unique values in the 'id' data set: {len(df['id'].unique())}")
```

```
Total rows in the data set: 569
Total count of unique values in the 'id' data set: 569
```

- diagnosis: Omit this column from the scaler because it is a classification column

I intend to use the Scaler library on the data except on ‘id’ and ‘diagnosis’.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
import numpy as np

stder = StandardScaler()
df = pd.read_csv("/Users/daniel/Desktop/Cancer_Data.csv")

df_scaled = df.copy(deep=True) # deep copy the df, just in case we want to see the original data
df_scaled = df_scaled.drop(columns='id')
df_scaled.iloc[:,1:] = stder.fit_transform(df_scaled.iloc[:,1:]) # scale columns 2 and forward

df_scaled['diagnosis'] = df_scaled['diagnosis'].replace('B',0).replace('M', 1) # encode the 'diagnosis' column
```

Plan for separating

First, I will separate my data for the columns with the measurements and a classification column. ‘x’ for the measurements, and ‘y’ for the classification.

```
x = df_scaled.iloc[:,1:].values
y = df_scaled['diagnosis'].values
```

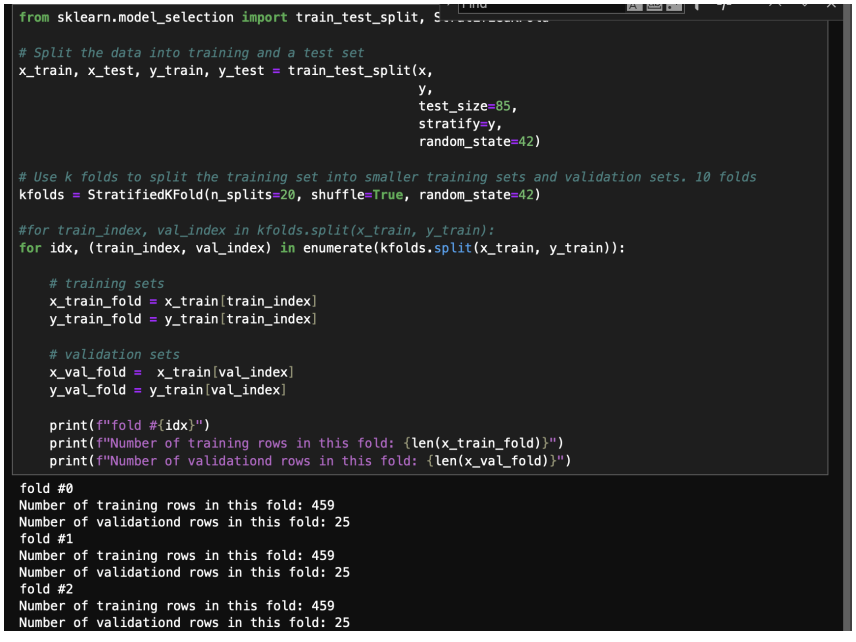
I then will use stratified K folds. Given my dataset has 569 rows, I will be more conservative with the amount of data separated for the test and validation sets. Instead of the typical 80/20 split, I will do an 85/15 split.

- Training Set: 70% (484 rows)
- Test Set: 15% (85 rows)

```
from sklearn.model_selection import train_test_split, StratifiedKFold

# Split the data into training and a test set
x_train, x_test, y_train, y_test = train_test_split(x,
                                                    y,
                                                    test_size=85,
                                                    stratify=y,
                                                    random_state=42)

print(f"Number of rows in the training set: {len(x_train)}")
print(f"Number of rows in the test set: {len(x_test)}")
```

```
Number of rows in the training set: 484
Number of rows in the test set: 85
```

I will then use 20 folds using the stratified K folds to further split my training data into smaller training sets and validation sets to use for hyperparameter tuning later on.



```
from sklearn.model_selection import train_test_split, S

# Split the data into training and a test set
x_train, x_test, y_train, y_test = train_test_split(x,
                                                    y,
                                                    test_size=85,
                                                    stratify=y,
                                                    random_state=42)

# Use k folds to split the training set into smaller training sets and validation sets. 10 folds
kfolds = StratifiedKFold(n_splits=20, shuffle=True, random_state=42)

#for train_index, val_index in kfolds.split(x_train, y_train):
for idx, (train_index, val_index) in enumerate(kfolds.split(x_train, y_train)):

    # training sets
    x_train_fold = x_train[train_index]
    y_train_fold = y_train[train_index]

    # validation sets
    x_val_fold =  x_train[val_index]
    y_val_fold = y_train[val_index]

    print(f"fold #{idx}")
    print(f"Number of training rows in this fold: {len(x_train_fold)}")
    print(f"Number of validationd rows in this fold: {len(x_val_fold)}")
```

```
fold #0
Number of training rows in this fold: 459
Number of validationd rows in this fold: 25
fold #1
Number of training rows in this fold: 459
Number of validationd rows in this fold: 25
fold #2
Number of training rows in this fold: 459
Number of validationd rows in this fold: 25
```

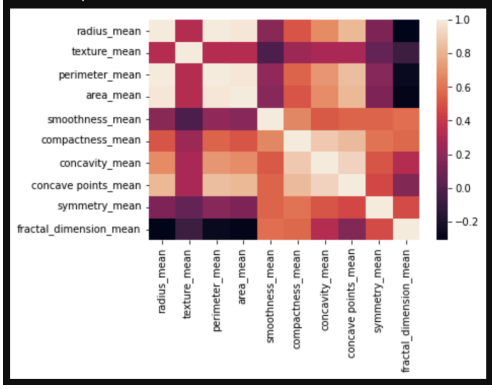
```
fold #18
Number of training rows in this fold: 460
Number of validationd rows in this fold: 24
fold #19
Number of training rows in this fold: 460
Number of validationd rows in this fold: 24
```

Visualization of Data:

First, I decided to visualize the mean, standard error, and worst columns separately. We can generally see the strongest correlations (negative included) are between the circle's sizing (radius, perimeter, area) vs the circle's shape (fractal_dimension, symmetry, concave points_mean). These relationships may offer insight into which columns can train the model better.

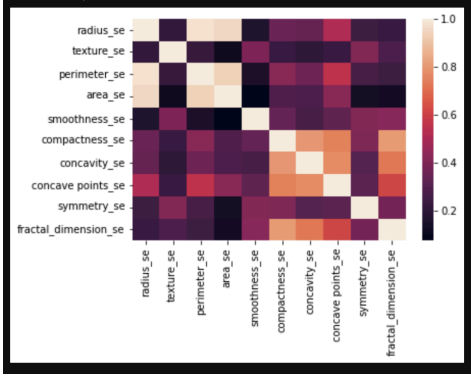
```
import seaborn as sns

mean_columns = ['radius_mean', 'texture_mean', 'perimeter_mean',
    'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
    'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean']

# mean columns
# represent feature correlation matrix using a heatmap
corr_mean = df[mean_columns].corr()

# plot the heatmap
sns.heatmap(corr_mean,
        xticklabels=corr_mean.columns,
        yticklabels=corr_mean.columns)
```

```
<AxesSubplot:>
```



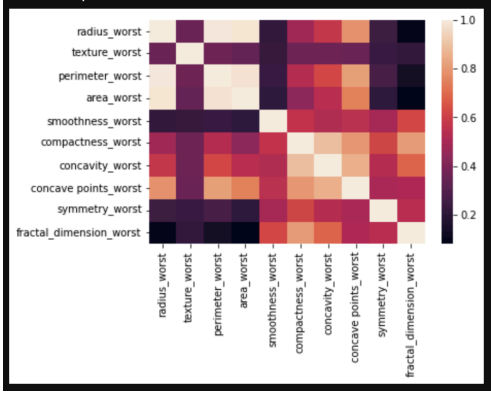

```
se_columns = ['radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
    'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
    'fractal_dimension_se']

# se columns
# represent feature correlation matrix using a heatmap
corr_se = df[se_columns].corr()

# plot the heatmap
sns.heatmap(corr_se,
        xticklabels=corr_se.columns,
        yticklabels=corr_se.columns)
```

```
<AxesSubplot:>
```



```
worst_columns = ['radius_worst', 'texture_worst',
    'perimeter_worst', 'area_worst', 'smoothness_worst',
    'compactness_worst', 'concavity_worst', 'concave points_worst',
    'symmetry_worst', 'fractal_dimension_worst']

# se columns
# represent feature correlation matrix using a heatmap
corr_worst = df[worst_columns].corr()

# plot the heatmap
sns.heatmap(corr_worst,
        xticklabels=corr_worst.columns,
        yticklabels=corr_worst.columns)
```

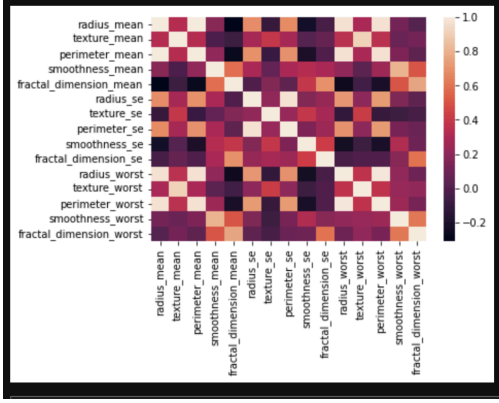
```
<AxesSubplot:>
```



Seeing these relationships, I took a mixture of columns related to the size vs shape of the cell's circle and created a heatmap out of those. Below, you can generally see the same correlations persist even across the ('mean' vs 'standard error') or' ('standard error' vs 'worst columns').

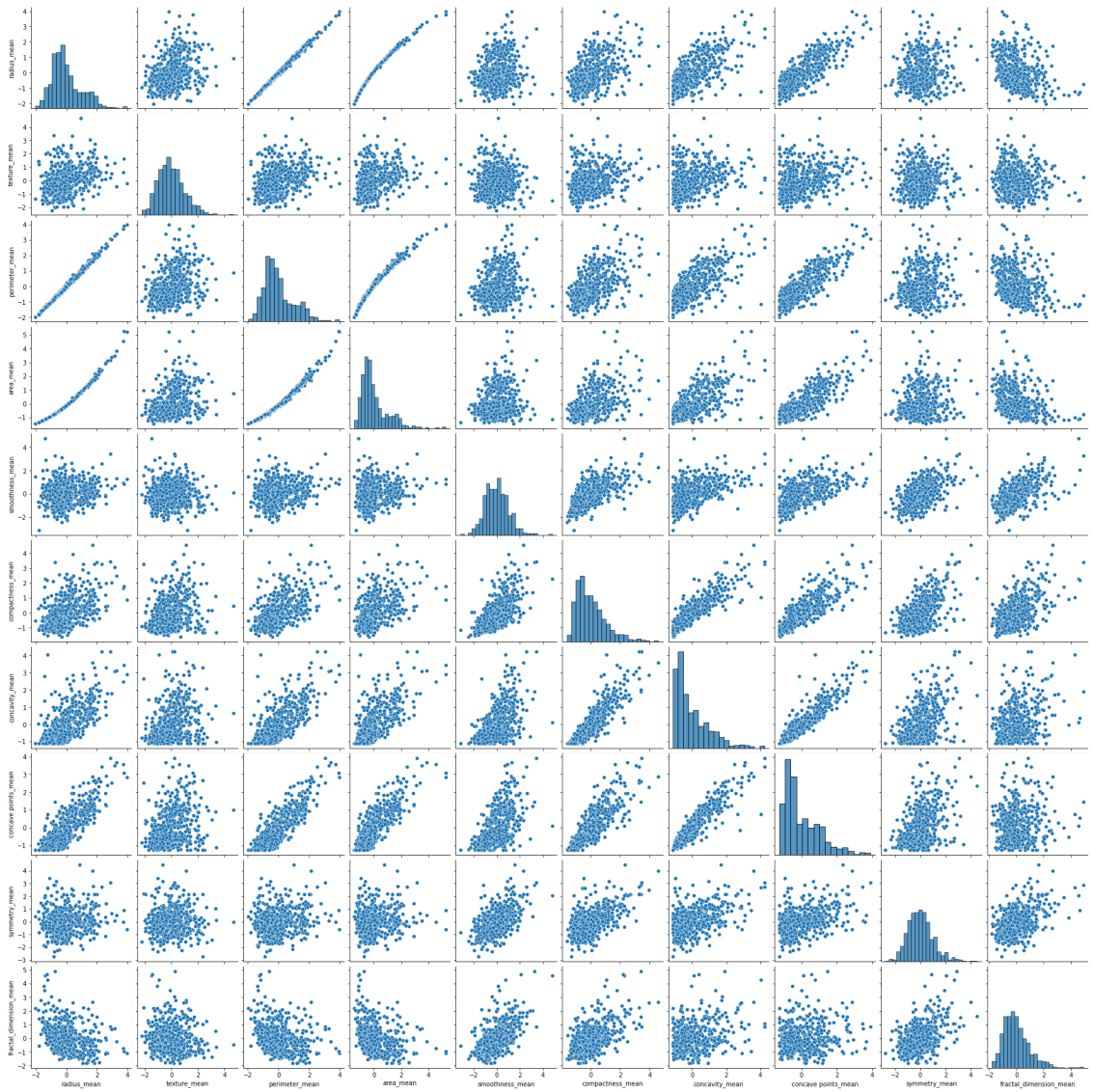
```
columns_mixed = [
    'radius_mean','texture_mean','perimeter_mean','smoothness_mean','fractal_dimension_mean',
    'radius_se','texture_se','perimeter_se','smoothness_se','fractal_dimension_se',
    'radius_worst','texture_worst','perimeter_worst','smoothness_worst','fractal_dimension_worst']

# represent feature correlation matrix using a heatmap
corr_mixed = df[columns_mixed].corr()

# plot the heatmap
sns.heatmap(corr_mixed,
        xticklabels=corr_mixed.columns,
        yticklabels=corr_mixed.columns)
```

```
<AxesSubplot:>
```



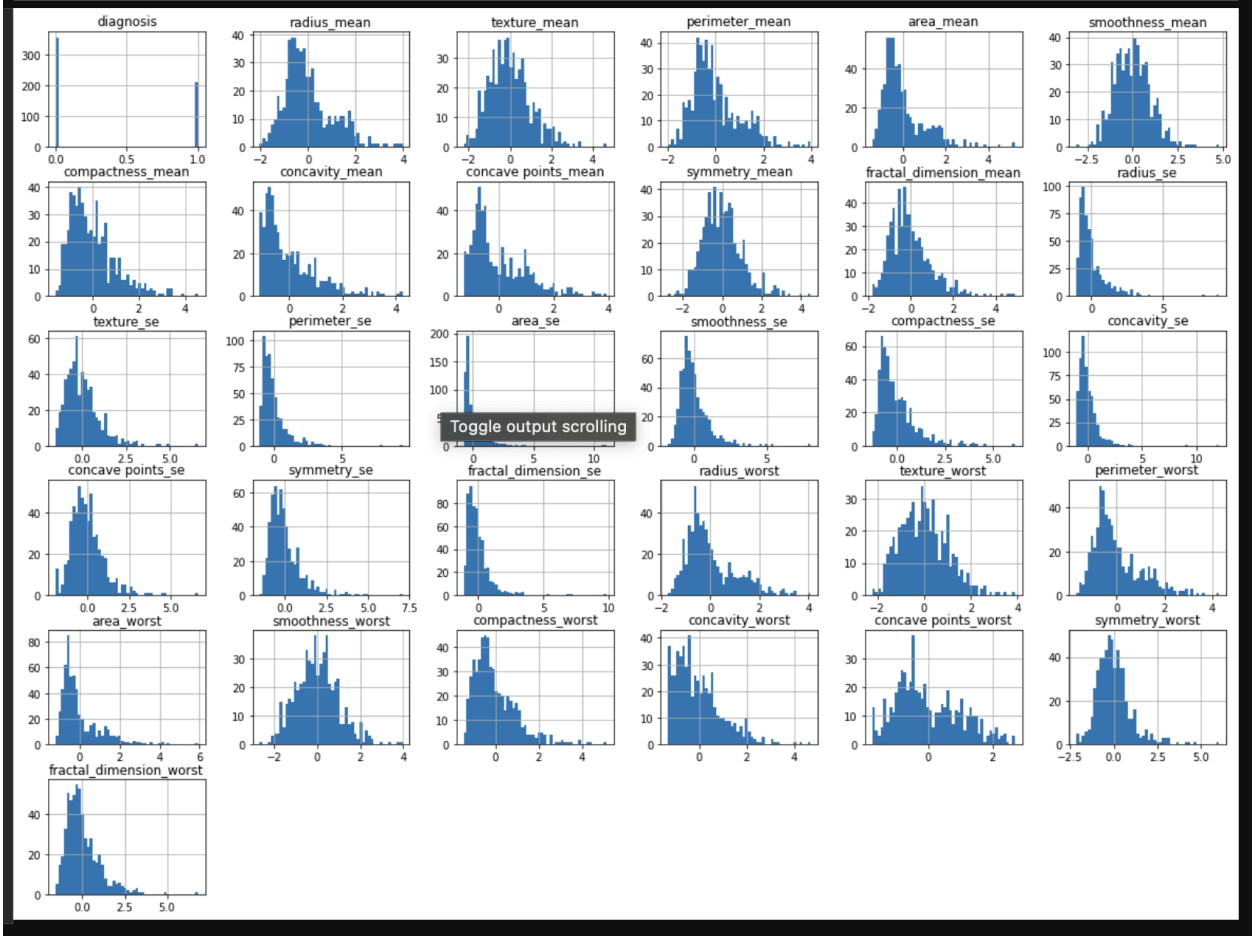
To provide emphasis on the correlations, I also created some scatter plots to better visualize the data. While there is a clear relationship between variables related to the size of the circle, we can also see clustering in the other plots. For example, we can see a cluster in the (‘perimeter_mean’ vs ‘symmetry_mean’). Because malignant tumors are abnormal in themselves, my intuition tells me that any points that fall outside of the cluster are indicative of a malignant diagnosis.

```
import seaborn as sns
import pandas as pd

# Pair plot of all columns
sns.pairplot(df_scaled[mean_columns])
plt.show()
```



I also plotted histograms for each field, and I can see all of the measurement columns follow a normal distribution or logarithmic distribution. Knowing every value is populated will offer reliability in the training models. Most evidently, there are no outliers visible in this data.

```
%matplotlib inline
import matplotlib.pyplot as plt
df_scaled.hist(bins=50, figsize=(20,15))
plt.show()
```



Part 3: Model Training

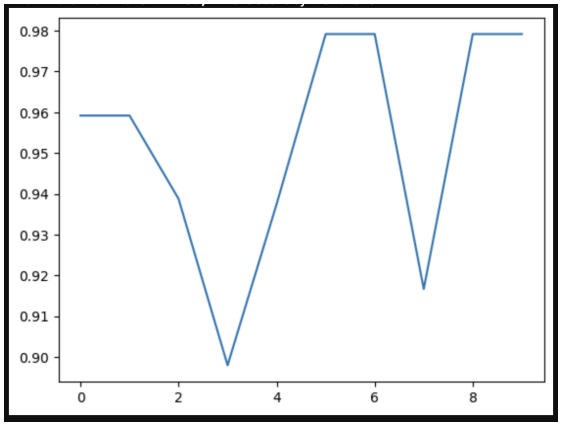
My three models include K-Nearest Neighbors (KNN), K Means, and Support Vector Machine (SVM).

Model 1: KNN

For my KNN model, the business objective is to capture our false positive rate and to see if we can improve it with Linear Discriminant Analysis (LDA). Because we have strong clustering evident in the scatterplots, KNN is an intuitive choice as my first model to train. To test multiple k values, I used the K-folds technique.

```
from sklearn.neighbors import KNeighborsClassifier

kfolds = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
k_choices = [1,2,3,4,8,10,12,15,20,50]
accuracies = []

#for train_index, val_index in kfolds.split(x_train, y_train):
for idx, (train_index, val_index) in enumerate(kfolds.split(x_train, y_train)):

    # training sets
    x_train_fold = x_train[train_index]
    y_train_fold = y_train[train_index]

    # validation sets
    x_val_fold =  x_train[val_index]
    y_val_fold = y_train[val_index]

    knn = KNeighborsClassifier(n_neighbors=k_choices[idx]) # choose a k_choice based on the value of i
    knn.fit(x_train_fold, y_train_fold)
    y_prediction = knn.predict(x_val_fold)

    num_correct = np.sum(y_prediction == y_val_fold) # number of matches found
    accuracy = float(num_correct) / len(y_val_fold) # calculate the accuracy
    accuracies.append(accuracy) # append our results
    print(f"For fold {idx+1} where k = {k_choices[idx]}, the accuracy is {accuracy:.4f}")

plt.plot(accuracies)
plt.show()
```

```
For fold 1 where k = 1, the accuracy is 0.9592
For fold 2 where k = 2, the accuracy is 0.9592
For fold 3 where k = 3, the accuracy is 0.9388
For fold 4 where k = 4, the accuracy is 0.8980
For fold 5 where k = 8, the accuracy is 0.9375
For fold 6 where k = 10, the accuracy is 0.9792
For fold 7 where k = 12, the accuracy is 0.9792
For fold 8 where k = 15, the accuracy is 0.9167
For fold 9 where k = 20, the accuracy is 0.9792
For fold 10 where k = 50, the accuracy is 0.9792
```

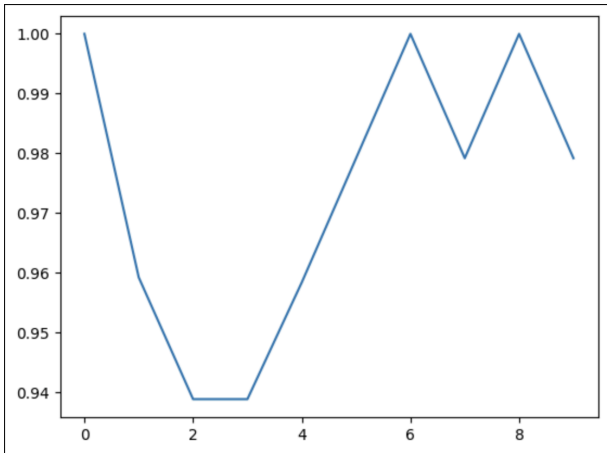


My results yielded accuracy results consistently >.90, but the best ones were k=10, 12, 20, and 50 at 97.92%. Normally this is a good score, but let's see if I can bring that up to 99% with dimensionality reduction. Since KNN is a supervised training model, hopefully LDA can improve the classification results through its ability to emphasize clustering. Again I use K folds here.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

lda = LDA(n_components=1) # chose 1 component since there are only 2 classes
x_train_lda = lda.fit_transform(x_train,y_train)
```

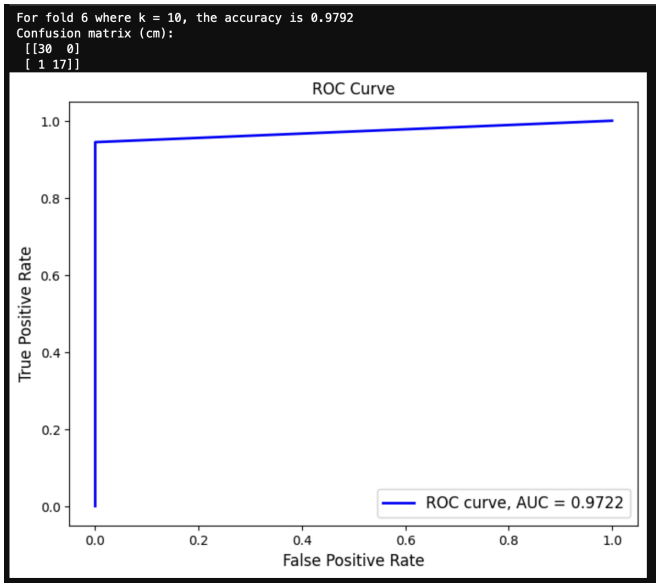
```
from sklearn.neighbors import KNeighborsClassifier

kfolds = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
k_choices = [1,2,3,4,8,10,12,15,20,50]
accuracies = []

#for train_index, val_index in kfolds.split(x_train, y_train):
for idx, (train_index, val_index) in enumerate(kfolds.split(x_train_lda, y_train)):

    # training sets
    x_train_fold = x_train_lda[train_index]
    y_train_fold = y_train[train_index]

    # validation sets
    x_val_fold =  x_train_lda[val_index]
    y_val_fold = y_train[val_index]

    knn = KNeighborsClassifier(n_neighbors=k_choices[idx]) # choose a k_choice based on the value of
    knn.fit(x_train_fold, y_train_fold)
    y_prediction = knn.predict(x_val_fold)

    num_correct = np.sum(y_prediction == y_val_fold) # number of matches found
    accuracy = float(num_correct) / len(y_val_fold) # calculate the accuracy
    accuracies.append(accuracy) # append our results
    print(f"For fold {idx+1} where k = {k_choices[idx]}, the accuracy is {accuracy:.4f}")

plt.plot(accuracies)
plt.show()
```

```
For fold 1 where k = 1, the accuracy is 1.0000
For fold 2 where k = 2, the accuracy is 0.9592
For fold 3 where k = 3, the accuracy is 0.9388
For fold 4 where k = 4, the accuracy is 0.9388
For fold 5 where k = 8, the accuracy is 0.9583
For fold 6 where k = 10, the accuracy is 0.9792
For fold 7 where k = 12, the accuracy is 1.0000
For fold 8 where k = 15, the accuracy is 0.9792
For fold 9 where k = 20, the accuracy is 1.0000
For fold 10 where k = 50, the accuracy is 0.9792
```



Our results show 1, 12, and 20 as 100%. These are likely overfitted models, so I will avoid these. However, k=10 has 97.92% in both the non-LDA and LDA models and is the highest accuracy recorded outside of the overfitted models. I will move forward with k=10.

Below I will calculate the accuracy, confusion matrix, and ROC curve results on one fold without using LDA data to avoid overfitting.

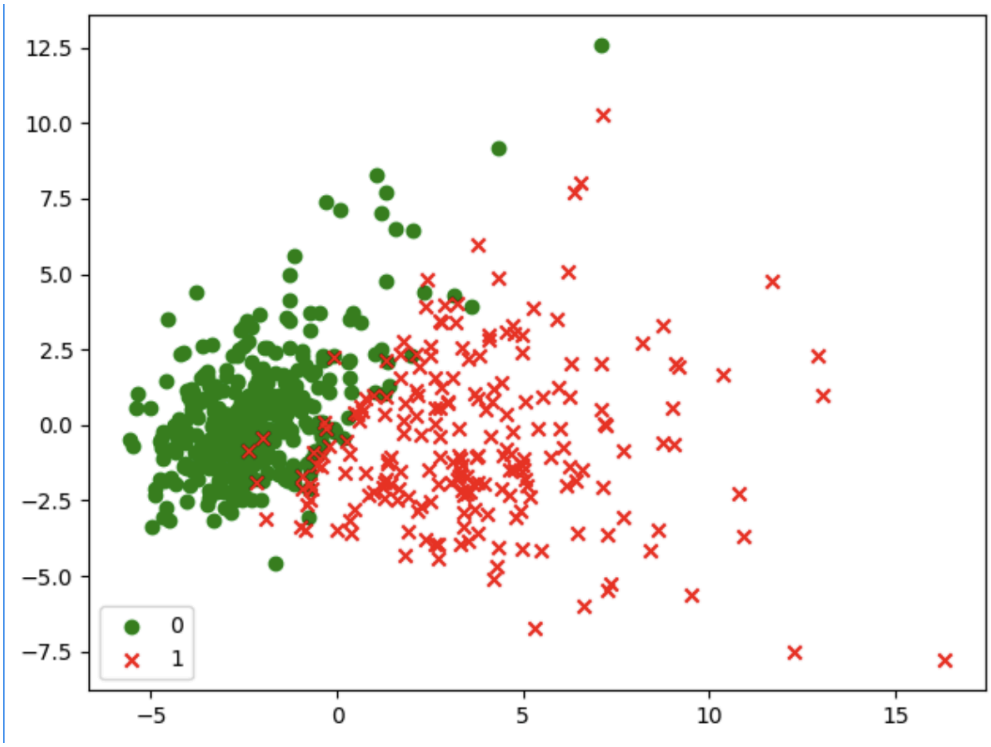


Our results show the Area Under the Curve (AUC) is .9722.
From my results from one fold, I can see the non-LDA data has an accuracy of 97.92.% where k=10. The non-LDA model even has no false negatives. My ROC curve is favorable with an "Area Under the Curve" (AUC) > .9, proving that KNN is a good classifier.

However in healthcare, any misdiagnosed cases can be deadly. 97.92% accuracy is good, but let's see if I can improve the accuracy in the following models.

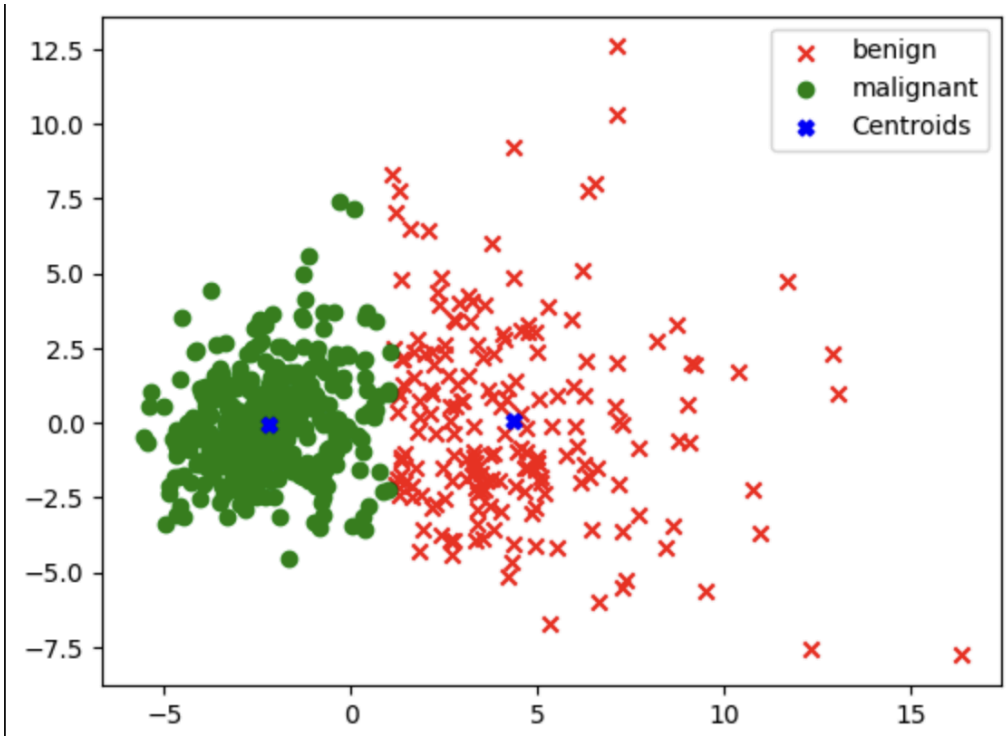
Model 2: K Means

The business objective here is to successfully cluster my data and prove the binary classifications can be separated by their features. Given the data has 30 features, I will use Principal Component Analysis (PCA) to shape the data into 2D. I can then use that plane to visualize the data and the feature separation. Afterwards, I would like to use KMeans to see how well the model clusters when using PCA data. Below is our clustering when using PCA on the data.



From my plot we can see the Benign cases (green circles) are more densely clustered and generally have an value < 5. Malignant cases (red x) are not as densely clustered, and also exhibit more variance. There is still a lot of overlap between the two classifiers however.

Having done PCA on our data, lets see if KMeans is able to properly cluster the data. Since I have a binary classifier, I know the number of my clusters will be 2. n_clusters=2



In the plot above, I can see that the clustering after using K Means seems pretty well defined. The blue x's are centroids, and one can see a clear boundary line between the two clusters. This indicates to me the feature space can be separated for malignant vs benign cases.

Model 3: Support Vector Machines

Having seen our features can be clustered definitively in our PCA -> KMeans model, let's use Support Vector Machine to see if we can draw a boundary between the two classes. The objective here will be to train the best model we can knowing we have good clustering. I will use the Linear SVC model on the same PCA data from our KMeans model, since we know it shows better clustering. Again I will use the K folds technique, but for only 6 folds this time. The hyperparameter I will be testing is C, and I will be testing different multiples of ten.

```
kfolds = StratifiedKFold(n_splits=6, shuffle=True, random_state=42)
C = [.001, .01, .1, 1, 10, 100]
accuracies = []

# split the x_pca data into training and test sets
x_train_pca, x_test_pca, y_train, y_test = train_test_split(x_pca,
                                                    y,
                                                    test_size=85,
                                                    stratify=y,
                                                    random_state=42)

#for train_index, val_index in kfolds.split(x_train, y_train):
for idx, (train_index, val_index) in enumerate(kfolds.split(x_train_pca, y_train)):

    # training sets
    x_train_fold = x_train[train_index]
    y_train_fold = y_train[train_index]

    # validation sets
    x_val_fold =  x_train[val_index]
    y_val_fold = y_train[val_index]

    svm = LinearSVC(C=C[idx]) # choose a k_choice based on the value of idx
    svm.fit(x_train_fold, y_train_fold)
    y_prediction = svm.predict(x_val_fold)

    num_correct = np.sum(y_prediction == y_val_fold) # number of matches found
    accuracy = float(num_correct) / len(y_val_fold) # calculate the accuracy
    accuracies.append(accuracy) # append our results
    print(f"For fold {idx+1} where C = {C[idx]}, the accuracy is {accuracy:.4f}")
```

```
For fold 1 where C = 0.001, the accuracy is 0.9630
For fold 2 where C = 0.01, the accuracy is 0.9259
For fold 3 where C = 0.1, the accuracy is 0.9753
For fold 4 where C = 1, the accuracy is 0.9630
For fold 5 where C = 10, the accuracy is 0.9875
For fold 6 where C = 100, the accuracy is 1.0000
```

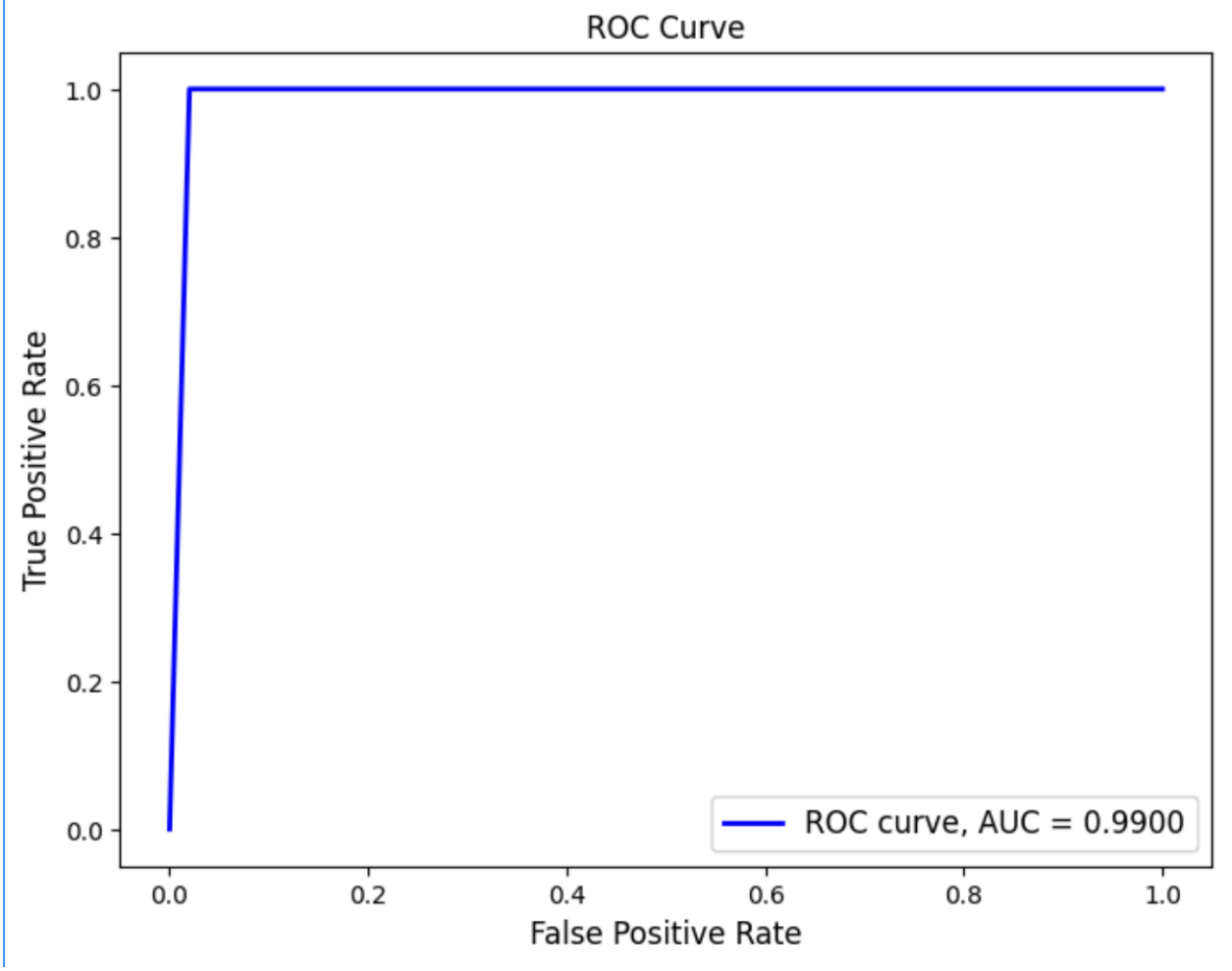
I can see from my accuracy results in the k folds that 10 may have the best fit. C=100 is likely overfitted at 100%. Using C=10 on one fold, I will plot the confusion matrix and calculate the ROC.

I can see from my accuracy results in the k folds that 10 may have the best fit. C=100 is likely overfitted at 100%. Using C=10 on one fold, I will plot the confusion matrix and calculate the ROC.

I can see above that the ROC curve is with an AUC=.9900. The model also has a 98.75% accuracy and no false negatives. A slightly better model compared to our KNN model at 97.92%

accuracy. These measurements indicate that SVM is a better classifier than KNN.

```
For fold 5 where C = 10, the accuracy is 0.9875
Confusion matrix (cm):
 [[49  1]
 [ 0 30]]
```



I can see above that the ROC curve is with an AUC=.9900. The model also has a 98.75% accuracy and no false negatives. A slightly better model compared to our KNN model at 97.92% accuracy. These measurements indicate that SVM is a better classifier than KNN. Despite the one false negative, SVM looks to be the best model to use for this data.

Part 4: Further Model Training

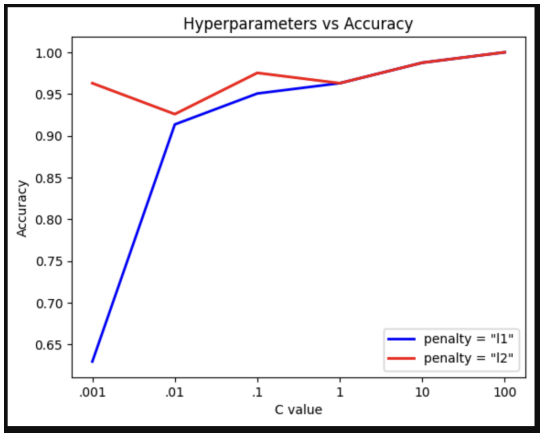
Next, I will expand on my Support Vector Machine (SVM) model and boost it with a K Means model. To begin, I only tested the C hyperparameter for my SVM model after the data was dimensionally reduced with Principal Component Analysis (PCA). I would like to see how well it performs using the additional "penalty" hyperparameter.

I will use the data that has been dimensionally reduced by PCA, "x_train_pca", and compare the accuracies with different "C" and "penalty" values. The hyperparameters will be tested with K folds.

```
# This time I will use 12 folds, because there are 12 combinations between my C and penalty values.
kfolds = StratifiedKFold(n_splits=6, shuffle=True, random_state=42)
C = [.001, .01, .1, 1, 10, 100]
penalties = ['l1','l2']
accuracies = []

#for train_index, val_index in kfolds.split(x_train, y_train):
for idx, (train_index, val_index) in enumerate(kfolds.split(x_train_pca, y_train)):

    # training sets
    x_train_fold = x_train[train_index]
    y_train_fold = y_train[train_index]

    # validation sets
    x_val_fold =  x_train[val_index]
    y_val_fold = y_train[val_index]

    for penalty in penalties:

        svm = LinearSVC(C=C[idx], penalty=penalty) # choose a k_choice based on the value of idx
        svm.fit(x_train_fold, y_train_fold)
        y_prediction = svm.predict(x_val_fold)

        num_correct = np.sum(y_prediction == y_val_fold) # number of matches found
        accuracy = float(num_correct) / len(y_val_fold) # calculate the accuracy
        accuracies.append(accuracy) # append our results
        print(f"For fold {idx+1} where C = {C[idx]} and penalty = {penalty}, the accuracy is {accuracy:.4f}")
```

```
For fold 1 where C = 0.001 and penalty = l1, the accuracy is 0.6296
For fold 1 where C = 0.001 and penalty = l2, the accuracy is 0.9630
For fold 2 where C = 0.01 and penalty = l1, the accuracy is 0.9136
For fold 2 where C = 0.01 and penalty = l2, the accuracy is 0.9259
For fold 3 where C = 0.1 and penalty = l1, the accuracy is 0.9506
For fold 3 where C = 0.1 and penalty = l2, the accuracy is 0.9753
For fold 4 where C = 1 and penalty = l1, the accuracy is 0.9630
For fold 4 where C = 1 and penalty = l2, the accuracy is 0.9630
For fold 5 where C = 10 and penalty = l1, the accuracy is 0.9875
For fold 5 where C = 10 and penalty = l2, the accuracy is 0.9875
For fold 6 where C = 100 and penalty = l1, the accuracy is 1.0000
For fold 6 where C = 100 and penalty = l2, the accuracy is 1.0000
```

My results yielded varying accuracies for each hyperparameter tuning, so I plotted them for a better visualization.



From my plot, I can see the accuracies generally performed better when penalty = "l2" over 6 different C values, so I will use "l2" for penalty. For C, I can see C=100 is 100% which means it's likely overfitted. Rather, I will use C=10 where both "l1" and "l2" showed 98.75% accuracy.

Next, I will boost the SVC model with results from my K Means model. In my KMeans model, I will measure the distance to the centroids for each point, and add it as a feature to my dataset before using SVM. This can hopefully improve my results.

```
# km object is from KMeans model
cluster_distances = km.transform(x_pca) # distances to each cluster centroid

x_pca_boosted = np.hstack((x_pca, cluster_distances)) # add distances as new features to the data

# Split the x_pca_boosted into training and test sets
x_pca_boosted_train, x_pca_boosted_test, y_train, y_test = train_test_split(x_pca_boosted,
                                                    y,
                                                    test_size=85,
                                                    stratify=y,
                                                    random_state=42)

print(f"training data size before boosting: {x_pca.shape}")
print(f"training data size before boosting: {x_pca_boosted.shape}")
```

```
training data size before boosting: (569, 2)
training data size before boosting: (569, 4)
```

With my new training data boosted, I will now fit the SVC model using penalty="l2" and C=10. I will then perform a prediction on the test data.

My data shape before boosting was 569 rows and two columns. After boosting it with results from my K Means model, it came out to 569 rows and four columns. With my new training data boosted, I fit the SVC model using penalty="l2" and C=10. I will then perform a prediction on the test data.

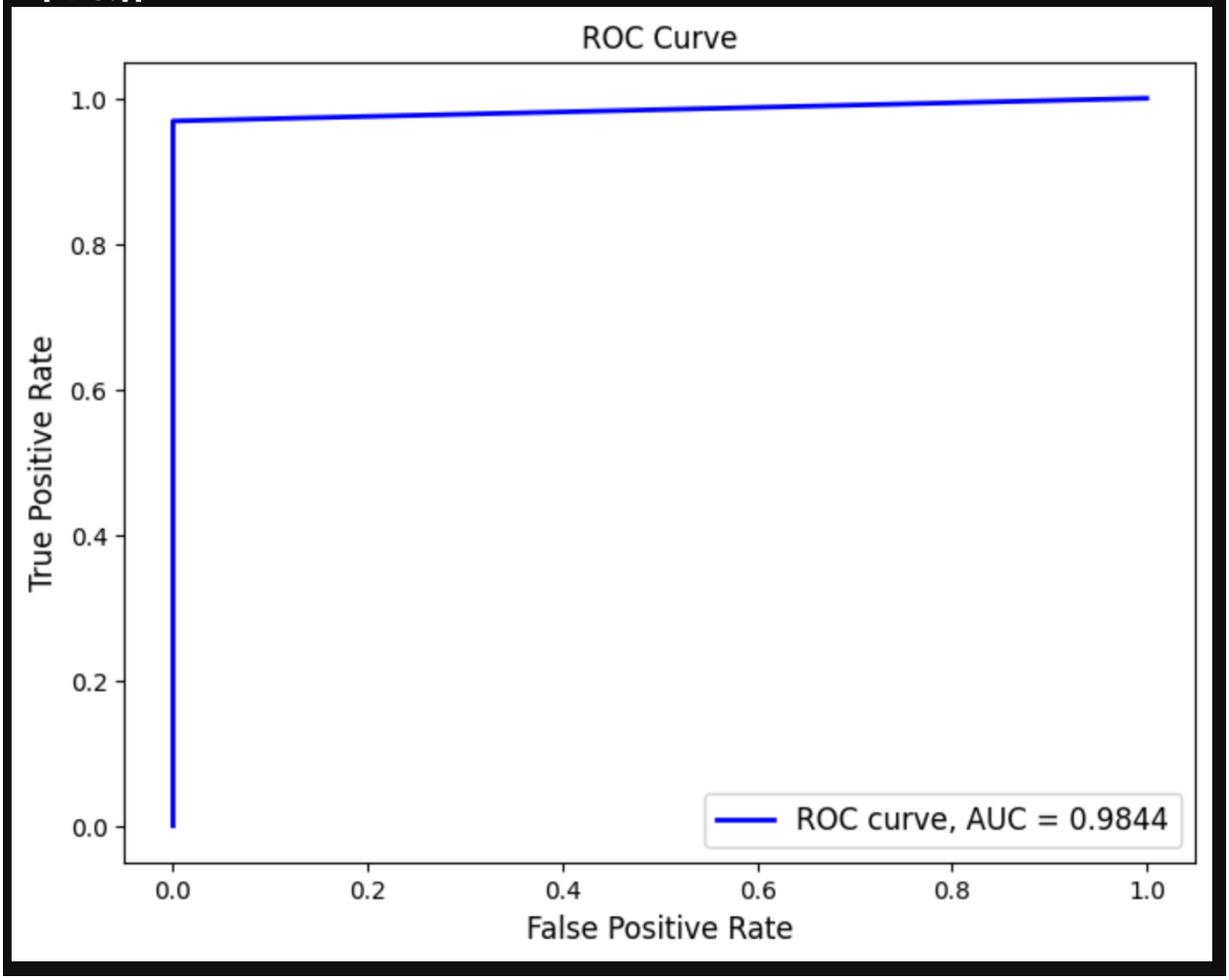
```
# fitting the SVC model with C=10 and penalty="l2"
svm = LinearSVC(C=10, penalty="l2")
svm.fit(x_pca_boosted_train, y_train)

y_prediction = svm.predict(x_pca_boosted_test) # prediction on the test data

num_correct = np.sum(y_prediction == y_test) # number of matches found
accuracy = float(num_correct) / len(y_test) # calculate the accuracy
accuracies.append(accuracy) # append our results
print(f"After PCA, and boosting from the K Means model, the SVM model with hyperparameters C = 10 and penalty = 'l2' yielded an accuracy of {accuracy:.4f}")
```

```
After PCA, and boosting from the K Means model, the SVM model with hyperparameters C = 10 and penalty = 'l2' yielded an accuracy of 0.9882
```

My accuracy came out to 98.82%. A desirable score for predicting benign vs malignant cases.

To further validate my results, I calculated the ROC and AUC. With an AUC of .9844, I know my model is reliable.

```
Confusion matrix (cm):
 [[53  0]
 [ 2 30]]
```



For future development, more data samples from a variety of sources would be helpful. The models yielded high accuracy, but the robustness against other data sets is yet to be determined. On a more technical note, this model could benefit from a bagging technique with three different models. Because only two of my models yielded binary classification results, I opted for a boosting technique instead. This proved successful with a 98.82 accuracy and no false negatives.

Part 5: Conclusion

My objective with this analysis was to create a reliable model to predict whether a case of breast cancer is benign or malignant. I also wanted to reduce the false negative rate due to how dangerous it would be to be declared cancer-free but still be malignant. My variables in the dataset are measurements about the mean, standard error, and worst values of characteristics of these cancer cells, as well as the classifier of benign vs malignant. After data normalization, I used three models to study the data:

- K Nearest Neighbors (KNN)
- K Means
- Support Vector Machines (SVM)

My goal with KNN was to see how well I could classify the data right away and use this weak predictor to tune my Linear Discriminant Analysis (LDA). By performing K folds on KNN ten times without LDA and ten times with LDA, I was able to discern that LDA data was not a good option due to a lot of overfitting. Without using LDA I was able to achieve a 97.92% accuracy using K=10 and an AUC score of .9722. A good score overall on a validation fold, but let's see if we can improve on this.

In my K Means model, my goal was to see if the features in the data can be separated. If the clustering of the data is strong enough, a model may be able to classify better on that data. In order to plot my data successfully, I used Principal Component Analysis (PCA) to transform the data to a 2D array and put emphasis on the clustering. The boundaries were not well drawn with a PCA transformation, but after using K Means on top of PCA the clustering came out with a definitive boundary line. This proved my binary classifier could be separated in a feature space.

In my third model, SVM, I tested how well clustering could occur with a linear boundary line. By using K folds to tune the hyperparameter, C, I was able to yield a 98.75% accuracy with an AUC of .99. These are good results, but this is only on training data on a fold. Seeing that K Means showed a boundary line and SVM proved to be a reliable predictor, I wanted to see if I can get a model that performs well with ensemble learning.

To improve my SVM model, I boosted the data with results from my K Means model. By taking the distance to the clusters' centroids as a feature, I appended that data to use for training in my SVM model. After fitting and predicting on the test data, I yielded 98.82% accuracy with an AUC score of .9844 and no false negatives.

In conclusion, my model successfully predicts whether a case of breast cancer is benign vs malignant when using the data with PCA, K Means, and then SVM. The model predicts at 99% rounded and has no false negatives, meeting my objective.