In [ ]:
```
# Python Libraries
# Pandas module works with the tabular data.
# Pandas has powerful tools like Series, DataFrame etc
# NumPy module works with numerical data.
# NumPy has a powerful tool like Arrays.
```

In [ ]:
```
import numpy as np
import pandas as pd
```

In [ ]:
```
# Loading Data Set
data = pd.read_csv('microcalcification.csv')
```

In [ ]:
```
# The head() function is used to get the first n rows
data.head()
# Target Variable or dependent variable = Microcalcification
# Independent Variable = Area,Grey Level,Gradient Strength,Noise Fluctuation,Contra
```

In [ ]:
```
# Total no of row and columns
data.shape
```

In [ ]:
```
# two classes -1 (no microcalcification) and  1(microcalcification)
data.Microcalcification.value_counts()
```

In [ ]:
```
# Removing Quotation
data['Microcalcification'] = data['Microcalcification'].str.replace("'", "")
```

In [ ]:
```
# Converting to integer data type
# Replacing -1 as 0
data['Microcalcification'] = data['Microcalcification'].astype(int)
data['Microcalcification'] = data['Microcalcification'].replace(-1, 0)
```

In [ ]:
```
data.head()
```

In [ ]:
```
# Count the number of NULL values in each column
print("Number of duplicates: ", data.isna().sum())
```

In [ ]:
```
# total Number of duplicates
print("Number of duplicates: ", data.duplicated().sum())
```

In [ ]:
```
#dropping duplicate values
df = data.drop_duplicates()
```

In [ ]:
```
#
df.Microcalcification.value_counts()
```

In [ ]:
```
# Seaborn is an amazing visualization library for statistical graphics plotting in
import seaborn as sns
import matplotlib.pyplot as plt
```

In [ ]:
```
# Data is imbalanced
# data points belonging to one single class variable dominating the other class
```

In [ ]:
```
grouped = df['Microcalcification'].value_counts()
categories = grouped.index
sizes = grouped.values
plt.figure(figsize = (10,6))
```

```
plt.pie(sizes, labels=categories, autopct='%1.1f%%', startangle=90)
plt.title('Microcalcification Class Distribution')
plt.axis('equal')
plt.show()
```

Interpretation: Data is imbalanced

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(13, 5))
sns.kdeplot(data=df['Area'], ax=axs[0], shade=True)
axs[0].set_title('Density Plot')
sns.kdeplot(data=df, x='Area', hue='Microcalcification', common_norm=False, fill=T
axs[1].set_title('PDF Plot')
plt.tight_layout()
plt.show()
```

In [ ]:
```
df['Area'].describe()
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(13, 5))
sns.kdeplot(data=df['Grey Level'], ax=axs[0], shade=True)
axs[0].set_title('Density Plot')
sns.kdeplot(data=df, x='Grey Level', hue='Microcalcification', common_norm=False,
axs[1].set_title('PDF Plot')
plt.tight_layout()
plt.show()
```

In [ ]:
```
df['Grey Level'].describe()
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(13, 5))
sns.kdeplot(data=df['Gradient Strength'], ax=axs[0], shade=True)
axs[0].set_title('Density Plot')
sns.kdeplot(data=df, x='Gradient Strength', hue='Microcalcification', common_norm=
axs[1].set_title('PDF Plot')
plt.tight_layout()
plt.show()
```

In [ ]:
```
df['Gradient Strength'].describe()
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(13, 5))
sns.kdeplot(data=df['Noise Fluctuation'], ax=axs[0], shade=True)
axs[0].set_title('Density Plot')
sns.kdeplot(data=df, x='Noise Fluctuation', hue='Microcalcification', common_norm=
axs[1].set_title('PDF Plot')
plt.tight_layout()
plt.show()
```

In [ ]:
```
df['Noise Fluctuation'].describe()
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(13, 5))
sns.kdeplot(data=df['Contrast'], ax=axs[0], shade=True)
axs[0].set_title('Density Plot')
sns.kdeplot(data=df, x='Contrast', hue='Microcalcification', common_norm=False, fi
axs[1].set_title('PDF Plot')
plt.tight_layout()
plt.show()
```

In [ ]:
```
df['Contrast'].describe()
```

In [ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(13, 5))
sns.kdeplot(data=df['Shape Descriptor'], ax=axs[0], shade=True)
```

```
axs[0].set_title('Density Plot')
sns.kdeplot(data=df, x='Shape Descriptor', hue='Microcalcification', common_norm=Fa
axs[1].set_title('PDF Plot')
plt.tight_layout()
plt.show()
```

In [ ]:
```
df['Shape Descriptor'].describe()
```

In [ ]:
```
#pair plot
# Which all feature is more important to classfying the data
x = df.drop('Microcalcification', axis=1)
y = df['Microcalcification']
df_plot = pd.concat([x, y], axis=1)
sns.set(style='ticks')
sns.pairplot(df_plot, hue='Microcalcification')
plt.show()
```

In [ ]:
```
#correlation
corr = df.corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
fig, ax = plt.subplots(figsize=(12, 10))
sns.heatmap(corr, mask=mask, annot=True, fmt=".2f", cbar=True, square=True, ax=ax)
ax.set_title("Correlation Heatmap")
plt.xlabel("Variables")
plt.ylabel("Variables")
plt.show()
```

In [ ]:
```
df.head()
```

In [ ]:
```
# Data Pre processing
```

In [ ]:
```
# Separating microcalcification from the data set X and Y
# Since the data in numeric format normalising the data
# Normalization is done to transform data into a single scale
# output of this is shown in a numpy array that is why its showing in a list
```

In [ ]:
```
from sklearn.preprocessing import StandardScaler

x = df.drop('Microcalcification', axis=1)
y = df['Microcalcification']

scaler = StandardScaler()
normalized_df = scaler.fit_transform(x)

print(normalized_df)
```

In [ ]:
```
# converting back from list to data frame
norm_df = pd.DataFrame(data=normalized_df, columns=("Area","Grey Level","Gradient S
norm_df.head()
```

In [ ]:
```
# Data is imbalanced so we are balancing the data by using SMOTE
# SMOTE its over sampling the data
# if we use down sampling then we will be left with only 500 data points
# since we need more data we are over sampling
```

In [ ]:
```
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
```

```
plt.title("Class Distribution Before Balancing")
plt.xlabel("Class Labels")
plt.ylabel("Count")
y.value_counts().plot(kind="bar")

smote = SMOTE(random_state=18)
x_cols, y_labels = smote.fit_resample(norm_df, y)

plt.subplot(1, 2, 2)
plt.title("Class Distribution After Balancing")
plt.xlabel("Class Labels")
plt.ylabel("Count")
y_labels.value_counts().plot(kind="bar")  # Plot y_labels instead of y
plt.tight_layout()
plt.show()
```

In [ ]:
```
# After overr sampling Total data in data set by adding synthetic data points using
print('total data points in the dataset after balancing:', x_cols.shape)
```

In [ ]:
```
# Splitting the training and testing data set
# Separated data set 75:25 train and test data
from sklearn.model_selection import train_test_split
```

In [ ]:
```
x_train, x_test, y_train, y_test = train_test_split(x_cols, y_labels, test_size=0.
print('number of samples in training data:', len(x_train))
print('number of samples in test data:', len(x_test))
```

In [ ]:
```
# Implementing Machine learning Models
```

Logistic Regression - L1 Penality

In [ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

In [ ]:
```
c = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]

tr_list = []
for i in c:
    clf = LogisticRegression(penalty='l1', tol=0.0001, C=i, fit_intercept=True, so
    clf.fit(x_train, y_train)
    pred = clf.predict(x_train)
    accuracy = accuracy_score(y_train, pred)
    tr_list.append(accuracy)
    print('Training accuracy at C:', i,'is', (accuracy*100), '%')
```

In [ ]:
```
optimal_index = np.argmax(tr_list)
optimal_c = c[optimal_index]
plt.figure(figsize=(10, 6))
plt.plot(c, tr_list, marker='o', linestyle='-', color='blue')
plt.scatter(optimal_c, tr_list[optimal_index], color='red', marker='o', label=f'Op
plt.axvline(x=optimal_c, color='gray', linestyle='--')
plt.xlabel('C Values')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. C Values')
plt.legend()
plt.xscale('log')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

In [ ]:
```
clf = LogisticRegression(penalty='l1', tol=0.0001, C=0.1, fit_intercept=True, solv
clf.fit(x_train, y_train)
pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, pred)
print('Test accuracy when L1 penality is used:', (accuracy*100), '%')
```

In [ ]:
```
print("\nFeature Importance:")
for feature, coef in zip(x_train.columns, clf.coef_[0]):
    print(f"{feature}: {coef}")
```

Logistic Regression - L2 Penality

In [ ]:
```
c = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]

tr_list = []
for i in c:
    clf = LogisticRegression(penalty='l2', C=i)
    clf.fit(x_train, y_train)
    pred = clf.predict(x_train)
    accuracy = accuracy_score(y_train, pred)
    tr_list.append(accuracy)
    print('Training accuracy at C:', i,'is', (accuracy*100), '%')
```

In [ ]:
```
optimal_index = np.argmax(tr_list)
optimal_c = c[optimal_index]
plt.figure(figsize=(10, 6))
plt.plot(c, tr_list, marker='o', linestyle='-', color='blue')
plt.scatter(optimal_c, tr_list[optimal_index], color='red', marker='o', label=f'Op
plt.axvline(x=optimal_c, color='gray', linestyle='--')
plt.xlabel('C Values')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. C Values')
plt.legend()
plt.xscale('log')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

In [ ]:
```
clf = LogisticRegression(penalty='l2', C=0.1)
clf.fit(x_train, y_train)
pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, pred)
print('Test accuracy when L2 penality is used:', (accuracy*100), '%')
```

In [ ]:
```
print("\nFeature Importance:")
for feature, coef in zip(x_train.columns, clf.coef_[0]):
    print(f"{feature}: {coef}")
```

Linear Support Vector Classifier - L1 Penality

In [ ]:
```
from sklearn.svm import LinearSVC

c = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]

tr_list = []
for i in c:
    clf = LinearSVC(penalty='l1', loss='squared_hinge', dual=False, C=i)
    clf.fit(x_train, y_train)
    pred = clf.predict(x_train)
    accuracy = accuracy_score(y_train, pred)
```

```
    tr_list.append(accuracy)
    print('Training accuracy at C:', i, 'is', (accuracy * 100), '%')
```

In [ ]:
```
optimal_index = np.argmax(tr_list)
optimal_c = c[optimal_index]
plt.figure(figsize=(10, 6))
plt.plot(c, tr_list, marker='o', linestyle='-', color='blue')
plt.scatter(optimal_c, tr_list[optimal_index], color='red', marker='o', label=f'Op
plt.axvline(x=optimal_c, color='gray', linestyle='--')
plt.xlabel('C Values')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. C Values')
plt.legend()
plt.xscale('log')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

In [ ]:
```
clf = LinearSVC(penalty='l1', loss='squared_hinge', dual=False, C=0.01)
clf.fit(x_train, y_train)
pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, pred)
print('Test accuracy when L1 penality is used:', (accuracy*100), '%')
```

In [ ]:
```
print("\nFeature Importance:")
for feature, coef in zip(x_train.columns, clf.coef_[0]):
    print(f"{feature}: {coef}")
```

Linear Support Vector Classifier - L2 Penality

In [ ]:
```
c = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]

tr_list = []
for i in c:
    clf = LinearSVC(penalty='l2', loss='squared_hinge', dual=False, C=i)
    clf.fit(x_train, y_train)
    pred = clf.predict(x_train)
    accuracy = accuracy_score(y_train, pred)
    tr_list.append(accuracy)
    print('Training accuracy at C:', i, 'is', (accuracy * 100), '%')
```

In [ ]:
```
optimal_index = np.argmax(tr_list)
optimal_c = c[optimal_index]
plt.figure(figsize=(10, 6))
plt.plot(c, tr_list, marker='o', linestyle='-', color='blue')
plt.scatter(optimal_c, tr_list[optimal_index], color='red', marker='o', label=f'Op
plt.axvline(x=optimal_c, color='gray', linestyle='--')
plt.xlabel('C Values')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. C Values')
plt.legend()
plt.xscale('log')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

In [ ]:
```
clf = LinearSVC(penalty='l2', loss='squared_hinge', dual=False, C=0.01)
clf.fit(x_train, y_train)
pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, pred)
print('Test accuracy when L2 penality is used:', (accuracy*100), '%')
```

In [ ]:
```
print("\nFeature Importance:")
for feature, coef in zip(x_train.columns, clf.coef_[0]):
    print(f"{feature}: {coef}")
```

Ensemble Models

Random Forest

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

est = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 75, 100]

tr_list = []
for i in est:
    clf = RandomForestClassifier(n_estimators=i, random_state=18)
    clf.fit(x_train, y_train)
    pred = clf.predict(x_train)
    accuracy = accuracy_score(y_train, pred)
    tr_list.append(accuracy)
    print('Training accuracy at n_estimators:', i, 'is', (accuracy * 100), '%')
```

In [ ]:
```
optimal_index = np.argmax(tr_list)
optimal_est = est[optimal_index]

plt.figure(figsize=(10, 6))
plt.plot(est, tr_list, marker='o', linestyle='-', color='blue')
plt.scatter(optimal_est, tr_list[optimal_index], color='red', marker='o', label=f'(
plt.axvline(x=optimal_est, color='gray', linestyle='--')
plt.xlabel('Number of Estimators')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. Number of Estimators')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

In [ ]:
```
clf = RandomForestClassifier(n_estimators=45, random_state=18)
clf.fit(x_train, y_train)
pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, pred)
print('Test accuracy when n_estimators = 45:', (accuracy*100), '%')
```

In [ ]:
```
feature_importance = clf.feature_importances_
print('\nFeature Importance:')
for feature, importance in zip(x_train.columns, feature_importance):
    print(f'{feature}: {importance:.4f}')
```

Gradient Boosted Decision Tree

In [ ]:
```
from sklearn.ensemble import GradientBoostingClassifier

est = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 75, 100]

tr_list = []
for i in est:
    clf = GradientBoostingClassifier(n_estimators=i, random_state=18)
    clf.fit(x_train, y_train)
    pred = clf.predict(x_train)
    accuracy = accuracy_score(y_train, pred)
```

```
    tr_list.append(accuracy)
    print('Training accuracy at n_estimators:', i, 'is', (accuracy * 100), '%')
```

In [ ]:
```
optimal_index = np.argmax(tr_list)
optimal_est = est[optimal_index]

plt.figure(figsize=(10, 6))
plt.plot(est, tr_list, marker='o', linestyle='-', color='blue')
plt.scatter(optimal_est, tr_list[optimal_index], color='red', marker='o', label=f'(
plt.axvline(x=optimal_est, color='gray', linestyle='--')
plt.xlabel('Number of Estimators')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. Number of Estimators')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

In [ ]:
```
clf = GradientBoostingClassifier(n_estimators=100, random_state=18)
clf.fit(x_train, y_train)
pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, pred)
print('Test accuracy when n_estimators = 100:', (accuracy*100), '%')
```

In [ ]:
```
feature_importance = clf.feature_importances_
print('\nFeature Importance:')
for feature, importance in zip(x_train.columns, feature_importance):
    print(f'{feature}: {importance:.4f}')
```