1. Dataset

The dataset that the research is working with is the “Cuban breast cancer” dataset. The dataset contains 23 columns and over 1600 rows. A portion of the dataset is still inefficient and requires further processing.

1.1. Messy and Noisy Data

Columns such as “menopause” and “agefirst” are unnecessarily using the object datatype and contains string values which can be represented using an integer. Some instance of the data uses the value “No” instead of an integer value to represent patients who have not experienced what the question asked. There are other columns with messy and noisy data that has duplicate values that represent the same meaning or unnecessary suffixes that are appended to the end of the data that can confuse the machine learning model.

```
array(['No', '3 months', '1 month', '2 months', '4 months', '8 months',
       '6 months', '5 months', '7 months', '10 months', '9 months', 'No ',
       '1 month ', '12 months', '6', '2', '10', '8', '0', '7', '4', '3',
       '5', '12', '11', '18', '16', '36', '15', '1', '72', '13', '24',
       '9', '48', '14', '17', '22', '26', '25', '28', '21'], dtype=object)
```

Figure 1 Values of the "breastfeeding" column

1.2. Instances with Multiple Values Assigned

Columns such as “nrelbc” and “allergies” contain a few rows that are assigned multiple values by appending additional values to a string separated by a slash. Since the machine learning model may not recognize rows with multiple values assigned in that way, the machine learning model may see it as a completely new value instead of two values being assigned to an instance which could confuse the machine learning model.

```
array(['Mother', 'Mother/Sister', 'Sister', 'Daughter', 'No',
       'Mother/Daughter', 'Sister/Daughter', 'Cousin', 'Aunt',
       'Grandmother', 'Mother/Grandmother', 'Aunt/Cousin',
       'Grandmother/Aunt', 'Sister/Grandmother', 'Grandmother ',
       'Mother/Aunt', 'Mother/Aunt/Cousin',
       'Mother/Grandmother/Aunt/Cousin'], dtype=object)
```

Figure 2 Values of the "nrelbc" column

1.3. Missing Data

A few columns in the dataset contains instances of data that are missing. Some columns contain only a few missing data populating only around 10 instances of data while other populate more than 500 instances of data.

id	0
age	0
menarche	0
menopause	0
agefirst	0
children	0
breastfeeding	0
nrelbc	0
biopsies	1
hyperplasia	0
race	0
year	537
imc	7
weight	10
exercise	0
alcohol	0
tobacco	0
allergies	276
emotional	0
depressive	0
histologicalclass	537
birads	537
cancer	0

Figure 3 Amount of missing values in each column

2. Dataset Preprocessing

To handle all the problems stated in the previous segment, the dataset was pre-processed before being utilized in any way. The relevant code used to preprocess the dataset can be found in the research's GitHub page on a file named "Data Preparation FINAL.ipynb". The results of the data preprocessing can be found in a csv file in the "Datasets" folder of the same GitHub page with the name "PreprocessedData.csv". For the raw unprocessed dataset, it can also be found in the same folder with the name "CubanDataset.csv".

2.1. Cleaning Messy and Noisy Data

This segment will aim to clean messy and noisy data contained in columns such as: "menopause", "agefirst", "breastfeeding", and "exercise"

2.1.1. Menopause Column

The "menopause" column contains the value "No" using the string datatype to represent the patient's inexperience having a menopause. This "No" value can easily be replaced by an existing integer value 0 that can be used to represent the same meaning.

```
array(['No', '47', '42', '44', '46', '43', '51', '40', '41', '48', '39',
       '49', '45', '50', '37', '52', '38', '32', '0', '33', '55', '53',
       '35', '54', '30', '60', '56', '34', '36'], dtype=object)
```

Figure 4 Values of the "menopause" column

Once the “No” values are replaced, the column will no longer be needed as an object datatype since all the values are now in integers and all string values have been replaced with its integer representation.

```
# Changing menopause column's datatype into integer and replacing the "No" value with 0
df["menopause"] = df["menopause"].replace(["No"], 0).astype(int)
```

Figure 5 Code for processing "menopause"

2.1.2. Agefirst Column

The “agefirst” column has the exact same problem as the “menopause” column where the string value “No” is used inefficiently and the column is casted as an object datatype.

```
array(['No', '36', '26', '21', '16', '20', '27', '24', '30', '35', '25',
       '18', '28', '23', '22', '17', '29', '19', '31', '14', '34', '33',
       '39', '32', '37', '38', '15', '0', '43', '46', '40', '10', '9'],
      dtype=object)
```

Figure 6 Values of the "agefirst" column

The solution to the problems of the “agefirst” column is also like the solution used in the “menopause” column.

```
# Changing agefirst column's datatype into integer and replacing the "No" value with 0
df["agefirst"] = df["agefirst"].replace(["No"], 0).astype(int)
```

Figure 7 Code for processing "agefirst"

2.1.3. Exercise Column

The “exercise” column contains duplicate values that represent the same meaning. The values “No”, “NO”, and integer value 0 all represent the same meaning. An instance of data in the column can also have the string value “Diary”.

```
array(['No', '2', '1', 'Diary', '3', '5', '4', 'NO', '6', '0', '7'],
      dtype=object)
```

Figure 8 Values of the "exercise" column

Instances of data that contains the value “No” and “NO” will be replaced with integer value 0 to ensure uniformity with previous processed data. Instances of data that contains the string value “Diary” will instead be replaced with 8. Once all string values are replaced with an integer value, the column will be casted as an integer datatype.

```
# Replacing the "No", and "NO" value with 0
df["exercise"] = df["exercise"].replace(["No", "NO"], 0)

# Replacing the "Diary" value with 8 and changing exercise column's datatype into integer
df["exercise"] = df["exercise"].replace("Diary", 8).astype(int)
```

Figure 9 Code for processing "exercise"

2.1.4. Breastfeeding Column

The “breastfeeding” column also has the same problems as the “exercise” column where there are duplicate values representing the same meaning in the form of string values “No”, “No ”, and integer value 0. In addition to that, some instances of data in the column contains unnecessary suffixes such as “-month” or “-months” appended to the values.

```
array(['No', '3 months', '1 month', '2 months', '4 months', '8 months',
       '6 months', '5 months', '7 months', '10 months', '9 months', 'No ',
       '1 month ', '12 months', '6', '2', '10', '8', '0', '7', '4', '3',
       '5', '12', '11', '18', '16', '36', '15', '1', '72', '13', '24',
       '9', '48', '14', '17', '22', '26', '25', '28', '21'], dtype=object)
```

Figure 10 Values of the "breastfeeding" column

The string values “No” and “No “will be replaced with integer value 0. Unnecessary suffixes will be removed using string replace functions alongside regex to help format the values. Once all the string values are replaced and all suffixes are removed, the column can be casted as an integer datatype.

```
# Cleaning the breastfeeding column's data
# Replacing the "No" and "No " value with 0
df["breastfeeding"] = df["breastfeeding"].replace(["No", "No "], 0)

# Removing month(s) string suffixes
df["breastfeeding"] = df["breastfeeding"].astype(str).str.replace(r" months?", "", regex = True) # (Operation converts datatype to string)

# Converting column datatype into integer
df["breastfeeding"] = df["breastfeeding"].astype(int)
```

Figure 11 Code for processing "breastfeeding"

2.2. Handling Data with Multiple Values Assigned

This segment will aim to improve the data representation for columns such as: “nrelbc” and “allergies”. Some values within these columns are assigned multiple values by appending more string values to the previous value separated by a slash. Since machine learning models will identify each instance of data in this form as its own unique value, this will negatively impact the ability of the machine learning model to predict labels. To solve this, each unique value available will get their own binary column to help identify what values are assigned to each instance of data.

2.2.1. Nrelbc Column

The “nrelbc” column is used to represent which relative or family member of the patient has previously suffered from breast cancer. The original state of the dataset still contains many instances of data with multiple values. Values such as “Sister/Grandmother” or “Aunt/Cousin” are some examples of instances of data that has multiple values assigned.

```
array(['Mother', 'Mother/Sister', 'Sister', 'Daughter', 'No',
       'Mother/Daughter', 'Sister/Daughter', 'Cousin', 'Aunt',
       'Grandmother', 'Mother/Grandmother', 'Aunt/Cousin',
       'Grandmother/Aunt', 'Sister/Grandmother', 'Grandmother ',
       'Mother/Aunt', 'Mother/Aunt/Cousin',
       'Mother/Grandmother/Aunt/Cousin'], dtype=object)
```

Figure 12 Values of the "nrelbc" column

Before creating a binary column, all available unique values must be identified first. The unique values in the "nrelbc" column are: "Mother", "Sister", "Daughter", "Cousin", "Aunt", "Grandmother", and "No". Then, create a binary column for each unique value available. To help identify whether each instance of data should have the value "True" or "False" for each binary column, use a loop that iterates through all available values and string functions that checks if a given string has another string within it. After all instances of data have been assigned a "True" or "False" value, drop the original column.

```
# Creating new columns to identify which relatives has experienced breast cancer before
# Identifying available values
familyMembers = ["Mother", "Sister", "Daughter", "Cousin", "Aunt", "Grandmother", "No"]

# Creating the new columns
for member in familyMembers:
    df[member] = df["nrelbc"].str.contains(member).astype(int)

# Dropping the original column
df.drop(columns = ["nrelbc"], inplace = True)
```

Figure 13 Code for processing "nrelbc"

2.2.2. Allergies Column

The "allergies" column has the same problem as the "nrelbc" column. However, before creating the binary columns for the "allergies" column, some missing data needs to be handled first. According to figure 3, the "allergies" column has about 276 missing data. Since this dataset is dealing with the patient's allergies, it can be assumed that any missing data are patients with no allergies. To handle this, all missing data will be replaced with the string value "No".

```
# Handling missing values in the allergies data
# Replacing missing values with the "No" value
df.fillna({"allergies" : "No"}, inplace = True)
```

Figure 14 Code for handling missing data in "allergies"

Once all missing data is handled with, the column should only have values that are significant to the dataset. The unique values in the "allergies" column are: "Rhinitis", "Medicines", "Laryngitis", "Dermatitis", "Other", and "No".

```
array(['Rhinitis', 'Medicines', 'Laryngitis', 'No', 'Dermatitis', 'Other',
       'Medicines/Other', 'Dermatitis/Rhinitis', 'Medicines/Rhinitis',
       'Medicines/Rhinitis/Laryngitis', 'Medicines/Rhinitis/Other',
       'Rhinitis/Laryngitis', 'Dermatitis/Rhinitis/Laryngitis',
       'Rhinitis/Other', 'Dermatitis/Rhinitis/Other',
       'Rhinitis/Laryngitis/Medicines', 'Laryngitis/Other',
       'Dermatitis/Rhinitis/Medicines', 'Rhinitis/Medicines',
       'Dermatitis/Medicines/Other', 'Laryngitis/Medicines',
       'Rhinitis/Laryngitis/Other', 'Dermatitis/Laryngitis',
       'Dermatitis/Rhinitis/Laryngitis/Medicines/Other'], dtype=object)
```

Figure 15 Values in the "allergies" column

Using the unique values, create a binary column for each value and assign a “True” or “False” value using loops and string functions. After the “True” or “False” values are assigned, the original column can be dropped.

```
# Creating new columns to identify which allergies exist in patients
# Identifying available values
allergies = ["Rhinitis", "Medicines", "Laryngitis", "Dermatitis", "Other", "No"]

# Creating the new columns
for allergy in allergies:
    df[allergy] = df["allergies"].str.contains(allergy).astype(int)

# Dropping the original column
df.drop(columns = ["allergies"], inplace = True)
```

Figure 16 Code for processing "allergies"

2.2.3. Renaming Resulting Binary Columns

To improve readability for the readers, it is recommended that each of the resulting binary columns be renamed to a more appropriate column name. In this case, the columns are renamed using the following code.

```
# Renaming the result columns
df = df.rename(columns = {
    "Mother" : "nrelbc_mother",
    "Sister" : "nrelbc_sister",
    "Daughter" : "nrelbc_daughter",
    "Cousin" : "nrelbc_cousin",
    "Aunt" : "nrelbc_aunt",
    "Grandmother" : "nrelbc_grandma",
    "No" : "nrelbc_absent"
})
```

```
# Renaming the result columns
df = df.rename(columns = {
    "Rhinitis" : "rhinitis_allergy",
    "Medicines" : "medicines_allergy",
    "Laryngitis" : "laryngitis_allergy",
    "Dermatitis" : "dermatitis_allergy",
    "Other" : "other_allergy",
    "No" : "no_allergy"
})
```

Figure 17 Code for renaming columns

2.3. Handling Missing Data with Low Significance

Columns such as “year”, “biopsies”, “imc”, and “weight” all have missing values. According to figure 3, the “year” column has about 537 missing data, but for the sake of this research, the “year” column will be deemed as low significance. Since the data inside the “year” column

isn't as significant, all missing data in the "year" column will be replaced with the median of all the values in the "year" column.

```
# Handling missing values in the year data
yearMedian = df["year"].median()
df.fillna({"year" : yearMedian}, inplace = True)
```

Figure 18 Code for processing "year"

Missing data in the "biopsies", "imc", and "weight" columns might possess crucial details that can be helpful for the machine learning model, but due to the few amount of missing data in these columns, rows that has missing data in these columns can be removed safely.

```
# Removing rows with missing values in all columns except for the histologicalclass and birads columns
df = df.drop(columns = ["histologicalclass", "birads"]).dropna().join(df["histologicalclass"]).join(df["birads"])
# Columns affected: "biopsies", "imc", "weight"
```

Figure 19 Code for removing missing data

In the code above, it should be noted that the columns "histologicalclass" and "birads" are exempted from the columns that are checked for missing data because these columns not only have a large amount of missing data, but the values can also be crucial to the machine learning model. These processing of these two columns will be handled in the next segment.

2.4. Handling Missing Values by Imputation

This segment aims to process the "histologicalclass" and "birads" column. These two columns possess a significant amount of missing data that can be crucial in training the machine learning model. This is why handling the missing data in these two columns are different to handling the missing data in the other columns. For imputation, the random forest regressor will be used to assign proper for the missing data.

2.4.1. Preparing Data for Imputation

Before imputation can be done, the dataset needs to be prepared first. All categorical data needs to be encoded beforehand. For this case, one hot encoding will be used to encode all the data except for the "birads" column which will be the target of the imputation later.

```
# Encoding categorical data using one hot encoding
oneHotChildren = pd.get_dummies(df["children"], drop_first = True)
oneHotHyperplasia = pd.get_dummies(df["hyperplasia"], drop_first = True)
oneHotRace = pd.get_dummies(df["race"], drop_first = True)
oneHotAlcohol = pd.get_dummies(df["alcohol"], drop_first = True)
oneHotTobacco = pd.get_dummies(df["tobacco"], drop_first = True)
oneHotEmotional = pd.get_dummies(df["emotional"], drop_first = True)
oneHotDepressive = pd.get_dummies(df["depressive"], drop_first = True)
oneHotCancer = pd.get_dummies(df["cancer"], drop_first = True)
```

Figure 20 Code for encoding data

This way of doing one hot encoding creates a new dataframe that has a certain number of binary columns in them depending on the unique values in each categorical column. All the new dataframe and the "birads" column will need to be rejoined back with the main dataframe.

```
# Selecting relevant non categorical features for imputation
relevantDf = df.select_dtypes(exclude = "object")
relevantDf["birads"] = df["birads"]

# Combining all relevant features and newly encoded data into one dataframe
preppedDf = pd.concat([relevantDf, oneHotChildren, oneHotHyperplasia, oneHotRace, oneHotAlcohol,
                       oneHotTobacco, oneHotEmotional, oneHotDepressive, oneHotCancer], axis = 1)
```

Figure 21 Code for rejoining all relevant dataframes

2.4.2. Histologicalclass Column

At this point in time while trying to handle the missing values in “histologicalclass”, the “birads” column still has missing values, which means that it isn’t viable to be used as an argument for the random forest regressor. For this step of imputing values into the “histologicalclass” column, the “birads” column will be dropped.

```
# Selecting required columns for imputation
histImputeDf = preppedDf.drop(["birads"], axis = 1)
```

Figure 22 Code for dropping "birads"

For random forest regressor to work, rows with missing “histologicalclass” values and rows without it needs to be separated. They will then be used as training and test sets for the random forest regressor.

```
# Splitting data into complete data and data with missing histologicalclass values
histTrainDf = histImputeDf.loc[~df.isnull().any(axis = 1)]
histTestDf = histImputeDf.loc[df.isnull().any(axis = 1)]
```

Figure 23 Code for splitting the data

Once the dataset has been split properly, the random forest regressor model can be trained with it and used to predict the missing values in the test set.

```
# Initializing the model
rf_hist = RandomForestRegressor()

# Selecting relevant features
features = histImputeDf.drop(columns = ["histologicalclass", "id"]).columns.tolist()

# Training the model
rf_hist.fit(histTrainDf[features], histTrainDf["histologicalclass"])
```

Figure 24 Code for training

```
# Prediciting the missing histologicalclass values
pred_hist = rf_hist.predict(histTestDf[features])
```

Figure 25 Code for predicting values

Since the results of the predictions come in the form of float type values. The results will be rounded to the nearest whole number.

```
# Rounding float values from prediction and changing the values to integer
pred_hist = pred_hist.round(0)
```

Figure 26 Code for rounding results

The values of the results will then be assigned to the test set, and the test set will be joined with the train set to create the whole dataset. The previously left out "birads" column will also be readded in this step. Since the data will be unsorted when joined, the "id" column will be used to help sort the data in ascending order based on their ID.

```
# Joining the results with the test dataframe
histTestDf.loc[:, "histologicalclass"] = pred_hist
```

Figure 27 Code for assigning value to the test set

```
# Joining the test dataframe with the training dataframe
newImputedDf = pd.concat([histTrainDf, histTestDf], ignore_index = True)
newImputedDf = newImputedDf.sort_values(by = "id", ascending = True)

# Rejoining the birads column that got dropped
newImputedDf["birads"] = preppedDf["birads"]
```

Figure 28 Code for rejoining all dataframes

2.4.3. Birads Column

Since the "histologicalclass" column has been processed and no more missing values can be found in that column, unlike when processing the "histologicalclass" column where we exclude the "birads" column, all the columns in the dataframe can be utilized to help impute the values of the "birads" column.

One additional step needs to be done before taking all the other steps needed to impute the missing values. The "birads" column is currently assigned as an object datatype. Each unique values of the "birads" column will need to be mapped out to an integer value for the random forest regressor to work.

```
# Encoding birads' categorical data
biradsMap = {
    "3A" : 1,
    "3B" : 2,
    "3C" : 3,
    "4B" : 4,
    "5B" : 5,
    "5C" : 6,
    "6" : 7
}
newImputedDf.loc[:, "birads"] = newImputedDf["birads"].map(biradsMap).fillna(0).astype(int)
```

Figure 29 Code for mapping "birads" value

Like the previous segment, once all the categorical values have been mapped to an integer value, the data will need to be split between rows with missing "birads" value and rows with complete data.

```
# Splitting data into complete data and data with missing birads values
birTrainDf = newImputedDf.loc[newImputedDf["birads"] != 0.0]
birTestDf = newImputedDf.loc[newImputedDf["birads"] == 0.0]
```

Figure 30 Code for splitting the data

Once the data has been split into their own dataframes, the random forest regressor model can be trained and used to predict the missing values.

```
# Initializing the model
rf_bir = RandomForestRegressor()

# Selecting relevant features
features = newImputedDf.drop(columns = ["birads", "id"]).columns.tolist()

# Training the model
rf_bir.fit(birTrainDf[features], birTrainDf["birads"])
```

Figure 31 Code for training

```
# Prediciting the missing histologicalclass values
pred_bir = rf_bir.predict(birTestDf[features])
```

Figure 32 Code for predicting values

The random forest regressor will return its prediction using the float datatype. The values will need to be rounded to its nearest whole number.

```
# Rounding float values from prediction and changing the values to integer
pred_bir = pred_bir.round(0)
```

Figure 33 Code for rounding

Once the values are rounded, the values will be assigned to the test set. The test set will then be joined with the train set and sorted using the values in the "id" column in ascending to order to form the final processed dataset.

```
# Joining the results with the test dataframe
birTestDf.loc[:, "birads"] = pred_bir
birTestDf.loc[:, "birads"] = birTestDf["birads"].astype(int)
```

Figure 34 Code for assigning value to the test set

```
# Joining the test dataframe with the training dataframe
FINAL_DF = pd.concat([birTrainDf, birTestDf], ignore_index = True)
FINAL_DF = FINAL_DF.sort_values(by = "id", ascending = True)
```

Figure 35 Code for rejoining all dataframes

3. Feature Selection

A portion of this research will be using a version of the dataset that only has features that are significant to the task. To identify which features will be used for this portion of the research, feature selection will be done on the pre-processed dataset. The code that was used for feature selection can be found on the research's GitHub page with the name "Feature Selection FINAL". The results of each feature selection method can be viewed inside the code.

For feature selection, three methods will be utilized. These methods include tree-based feature importance using random forest classifier, recursive feature elimination using logistic regression, and select k-best using chi2.

3.1. Preparing The Data

Before feeding the pre-processed data into the feature selection methods, some steps need to be taken. The necessary features will need to be selected, and the data will need to be split using the train test split function.

```
# Selecting non-label features
features = df.drop(columns = ["cancer", "id"]).columns.tolist()

# Train test split
X = df[features]
y = df["cancer"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

Figure 36 Code for preparing the data

One thing that should be noted is that since a train test split with a random state is being used, the output of each execution of each method will be different. Despite the randomness, a few of the most significant features should be able to consistently outscore other features on the list.

3.2. Tree-Based Feature Importance

For this method, the random forest classifier will be used. The model will be fitted on the training set and output the features that it deems significant to the task.

```
# Initializing the model
model1 = RandomForestClassifier()

# Model fitting
model1.fit(X_train, y_train)
importances = model1.feature_importances_
```

Figure 37 Random forest classifier model

According to the output of this model, the consistent top selected features are the columns “biopsies”, “histologicalclass”, “is_sad”, and “consumed_alcohol”

3.3. Recursive Feature Elimination

For this method, the logistic regression model will be used. The model will be fitted on the training set and asked to choose and output five of the most significant features.

```
# Initializing the model
model2 = LogisticRegression(max_iter = 9999)

# Model fitting
rfe = RFE(model2, n_features_to_select = 5)
X_train_rfe = rfe.fit_transform(X_train, y_train)
```

Figure 38 Logistic regression model

According to the output of this model, the consistently selected five top features are the columns “biopsies”, “other_allergy”, “2_children”, “3_children”, and “is_sad”

3.4. Select K-Best

For this method, the select k-best method will be used using the chi2 scoring function to find the top five most significant features.

```
# Applying select k-best using chi2
select_k_best = SelectKBest(score_func = chi2, k = 5)
X_train_k_best = select_k_best.fit_transform(X_train, y_train)
```

Figure 39 Select k-best model

According to the output of this model, the consistently selected top five features are the columns “age”, “menopause”, “agefirst”, “breastfeeding”, and “biopsies”

3.5. Features Selected for Research

Since all the methods used above have concluded different features, the features that will be used for the research are chosen manually after considering all the outputs of each model. The chosen features that will be used for this research are “biopsies”, “histologicalclass”, “consumed_alcohol”, “menopause”, and “is_sad”

4. Methods

This segment will explain the choice of machine learning models that will be used in this research and what method of evaluation will be used.

4.1. Problem Definition

This research’s main goal is to predict the existence of breast cancer using features available in the dataset. The target label itself only has the two possible values “True” or “False”. This means that the research will be focused on performing binary classification on the given dataset.

4.2. Models

The models that will be tested for this task and research are:

- Random Forest Classifier
- Logistic Regression
- K-Nearest Neighbour (KNN)
- Support Vector Machine (SVM)

Each of this model can be used for binary classification but works in different ways and has each of its own unique strengths making it suitable for this research.

4.3. Evaluation Metrics

To evaluate the above models, a few metrics will be used. These metrics are the accuracy score, precision score, recall score, and f1-score. Accuracy score is used to get a general measure on the model’s performance. Precision score is used to measure the accuracy of positive predictions. Recall score is used to measure the coverage of actual positives. F1-score is used to measure the balance between precision and recall scores.

5. Training, Testing, and Evaluating the Models

This segment will focus on training, testing, and evaluating the models whilst also making a comparison between all the models used that were being trained using which type of dataset. All relevant files that contain the codes for the models are included in the research’s GitHub page. Files with the model’s code in it will have the prefix “MODEL” in their names.

5.1. Train Test Split

Before feeding the dataset into the model, two train test splits will need to be created. One train test split will be used for the dataset using all the features available, and the other will be used for the dataset that uses only the chosen good features.

The train test split needs to have a valid feature set X and target variable y. For both train test splits, the target variable y will be the same being the "cancer" label. The good feature train test split will only include the good features chosen for its feature set X while the normal train test split will include all features except for the "cancer" label.

```
# Selecting the good featuers from feature selection
good_features = ["biopsies", "histologicalclass", "consumed_alcohol", "menopause", "is_sad"]
```

Figure 40 Code for selecting the good features

```
# Assigning the X and Y
X_normal = df.drop(columns = ["cancer"])
X_good = df[good_features]
y = df["cancer"]
```

Figure 41 Code for assigning X and y

Once all the feature set and target variable have been set, the train test split can be constructed. For this research, a test size of 30% will be used alongside random states.

```
# Creating the train test split for the experiment that's using the good features
X_good_train, X_good_test, y_good_train, y_good_test = train_test_split(X_good, y, test_size = 0.3, random_state = 42)
```

Figure 42 Code for creating the train test split

5.2. Model Training and Fitting

In general, how each model is trained and fitted in code are similar. The difference exists only when initializing the model and setting the model's parameters. The code between the two types of experiments being done on each model is also similar and only changes the variables created by the train test split so that the variable used are according to the dataset being experimented on. The parameters used in each model will be shown below.

For the random forest classifier model, this research will use 100 estimators or trees alongside a random state.

```
# Initializing and fitting the model
model1 = RandomForestClassifier(n_estimators=100, random_state=42)
model1.fit(X_normal_train, y_normal_train)
```

Figure 43 Code for initializing the random forest classifier

For the logistic regression model, this research will use a max iteration of 1000 alongside a random state.

```
# Initializing and fitting the model
model1 = LogisticRegression(max_iter=1000, random_state=42)
model1.fit(X_normal_train, y_normal_train)
```

Figure 44 Code for initializing the logistic regression model

For the KNN model, this research will have the model consider 5 of its nearest neighbours.

```
# Initializing and fitting the model
model1 = KNeighborsClassifier(n_neighbors=5)
model1.fit(X_normal_train, y_normal_train)
```

Figure 45 Code for initializing the KNN model

For the SVM SVC model, this experiment will use the "rbf" kernel for the algorithm, a regularization parameter C of 1.0 to help the model not overfit or generalize too much, and the "scale" setting for the gamma parameter to ensure that the gamma of the model is automatically adjusted based on the dataset and feature variance.

```
# Initializing and fitting the model
model1 = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
model1.fit(X_normal_train, y_normal_train)
```

Figure 46 Code for initializing the SVM SVC model

Once the model is initialized and fitted to the train split of each dataset, the model can start predicting the test sets' label of each dataset.

```
# Making predictions using the model
y_pred = model1.predict(X_normal_test)
```

Figure 47 Code for making predictions

It should be noted that all the figures above use the code for the experiment that uses the normal dataset that has all the available features. The code for the experiment that only uses the chosen good features will have similar code. In the other experiment's code, variable names such as "model1" will be changed to "model2", "X_normal_train" will be changed to "X_good_train", "y_normal_train" will be changed to "y_good_train", and some other variable names were also changed. To view the full code, visit the research's GitHub page.

5.3. Evaluating The Model

To evaluate the model, the research will use the accuracy score, precision score, recall score, and f1-score. The code will also provide a confusion matrix and supports metric to help provide additional information about the model's performance and distribution.

```
# Evaluating the model
accuracy = accuracy_score(y_normal_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

conf_matrix = confusion_matrix(y_normal_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

class_report = classification_report(y_normal_test, y_pred)
print("Classification Report:")
print(class_report)
```

Figure 48 Code for evaluating the model

6. Results

This segment will focus on the resulting performance of each model when being tested on each type of dataset and making a comparison between each experiment's results. The results of each experiment can also be viewed in the code itself.

6.1. Random Forest Classifier Evaluation

When using all the features, the random forest classifier managed to reach peak scores hitting 100% scores on all the metrics used. This is a possible indication of overfitting, possibly due to the existence of too much noisy data.

When using only the good features, the model managed to almost reach the scores of the other experiment but fell short in a lot of the metrics, only having 100% scores on two metrics. This could be an indication the overfitting isn't as bad as the other experiment.

```
Accuracy: 1.00
Confusion Matrix:
[[165   0]
 [  0 341]]
Classification Report:
              precision    recall  f1-score   support

       False       1.00      1.00      1.00       165
        True       1.00      1.00      1.00       341

    accuracy                           1.00       506
   macro avg       1.00      1.00      1.00       506
weighted avg       1.00      1.00      1.00       506
```

Figure 49 Scoring of experiment using all features

```
Accuracy: 0.99
Confusion Matrix:
[[165   0]
 [  6 335]]
Classification Report:
              precision    recall  f1-score   support

       False       0.96      1.00      0.98       165
        True       1.00      0.98      0.99       341

    accuracy                           0.99       506
   macro avg       0.98      0.99      0.99       506
weighted avg       0.99      0.99      0.99       506
```

Figure 50 Scoring of experiment using only good features

6.2. Logistic Regression Evaluation

The results of both experiments that were using logistic regression are like the results of the experiments using random forest classifier. The experiment using all the features managed to get perfect scores on all metrics while the experiment using the chosen good features only got perfect scores on a few metrics.

```
Accuracy: 1.00
Confusion Matrix:
[[165   0]
 [  0 341]]
Classification Report:
              precision    recall  f1-score   support

       False       1.00      1.00      1.00       165
        True       1.00      1.00      1.00       341

    accuracy                           1.00       506
   macro avg       1.00      1.00      1.00       506
weighted avg       1.00      1.00      1.00       506
```

Figure 51 Scoring of experiment using all features

```
Accuracy: 0.99
Confusion Matrix:
[[163   2]
 [  1 340]]
Classification Report:
              precision    recall  f1-score   support

       False       0.99      0.99      0.99       165
        True       0.99      1.00      1.00       341

    accuracy                           0.99       506
   macro avg       0.99      0.99      0.99       506
weighted avg       0.99      0.99      0.99       506
```

Figure 52 Scoring of experiment using only good features

6.3. K-Nearest Neighbour Evaluation

The results of the KNN experiments are also like the previous two experiments. Peak scores were reached on all metrics for the experiment using all the features, and an exceptionally high metrics score with a few perfect scores was reached for the experiment using the chosen good features.

```
Accuracy: 1.00
Confusion Matrix:
[[165   0]
 [  0 341]]
Classification Report:
              precision    recall  f1-score   support

       False       1.00      1.00      1.00       165
        True       1.00      1.00      1.00       341

    accuracy                           1.00       506
   macro avg       1.00      1.00      1.00       506
weighted avg       1.00      1.00      1.00       506
```

Figure 53 Scoring of experiment using all features

```
Accuracy: 0.99
Confusion Matrix:
[[164   1]
 [  3 338]]
Classification Report:
              precision    recall  f1-score   support

       False       0.98      0.99      0.99       165
        True       1.00      0.99      0.99       341

    accuracy                           0.99       506
   macro avg       0.99      0.99      0.99       506
weighted avg       0.99      0.99      0.99       506
```

Figure 54 Scoring of experiment using only good features

6.4. Support Vector Machine Evaluation

The result of the SVM experiment that uses all the available features received a near perfect metric score with a few metrics reaching 100% score, which is similar to previous experiments. The results of the experiment that uses the chosen good features is where it differs from previous experiments. The result of this experiment averages out on an 80% metric score with an accuracy of 83%.

```
Accuracy: 0.99
Confusion Matrix:
[[162   3]
 [  0 341]]
Classification Report:
              precision    recall  f1-score   support

       False       1.00      0.98      0.99       165
        True       0.99      1.00      1.00       341

    accuracy                           0.99       506
   macro avg       1.00      0.99      0.99       506
weighted avg       0.99      0.99      0.99       506
```

Figure 55 Scoring of experiment using all features

```
Accuracy: 0.83
Confusion Matrix:
[[ 97  68]
 [ 16 325]]
Classification Report:
              precision    recall  f1-score   support

       False       0.86      0.59      0.70       165
        True       0.83      0.95      0.89       341

    accuracy                           0.83       506
   macro avg       0.84      0.77      0.79       506
weighted avg       0.84      0.83      0.82       506
```

Figure 56 Scoring of experiment using only good features