University of Málaga

Intelligent Systems

Scikit-Learn assignment

Author:
Mario Pascual González

Assigned Professors:
Dr. Ezequiel López Rubio
Dr. Enrique Domínguez Merino

1 Dataset introduction

The Breast Cancer Winsconsin (BCW) dataset[1] was created by Dr. William H. Wolberg at the University of Wisconsin, and it aims to predict whether a breast mass is benign or malignant based on various features obtained from a digitized image of a fine needle aspirate (FNA) of a breast mass.

The dataset consists of 30 real-valued features computed from a digitized image of a breast mass. These features describe the characteristics of the cell nuclei present in the image.

The 30 features are actually 10 different metrics each computed in 3 different ways (mean, standard error, and worst/largest).

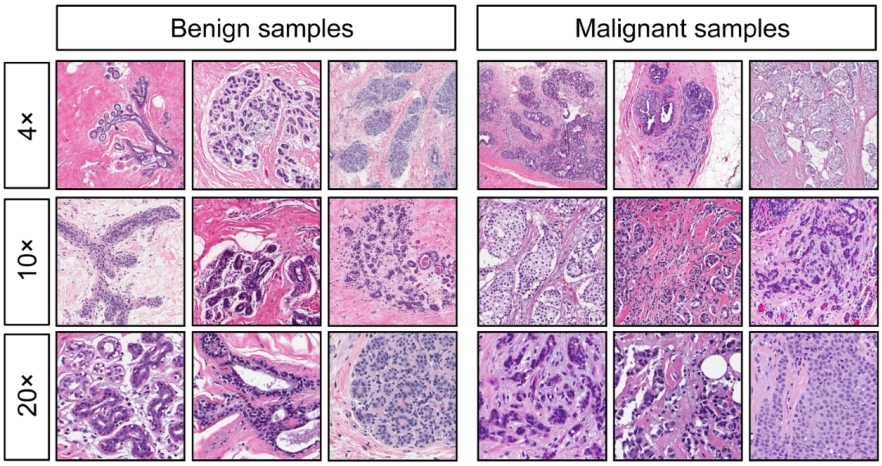

Figure 1: FNA of breast mass samples from the BCW dataset

radius_mean	texture_mean	perimeter_mean	area_mean	(...)	target
18.0	10.4	122.8	1001.0	(...)	1
20.6	17.8	132.9	1326.0	(...)	1
19.7	21.2	130.0	1203.0	(...)	1
11.4	20.4	77.6	386.1	(...)	1
20.3	14.3	135.1	1297.0	(...)	1
12.4	15.7	82.6	477.1	(...)	1

Table 1: Peek at the first columns and rows of the dataset

1.1 Feature description

Feature	Description
id	Identifier to uniquely distinguish each record
radius_mean	Mean distance from the center of the cell nucleus to its boundary
texture_mean	Mean standard deviation of the grayscale intensity values
perimeter_mean	Mean distance around the cell nucleus
area_mean	Mean size of the nucleus
smoothness_mean	Mean local variation in radius lengths
compactness_mean	Mean compactness, calculated as $\frac{Perimeter^2}{Area-1}$
concavity_mean	Mean severity of concave portions on the boundary
concave points_mean	Mean number of concave portions on the boundary
symmetry_mean	Mean symmetry of the nucleus
fractal_dimension_mean	Mean "coastline approximation" or fractal dimension
radius_se	Standard error of radius
texture_se	Standard error of texture
perimeter_se	Standard error of perimeter
area_se	Standard error of area
smoothness_se	Standard error of smoothness
compactness_se	Standard error of compactness
concavity_se	Standard error of concavity
concave points_se	Standard error of concave points
symmetry_se	Standard error of symmetry
fractal_dimension_se	Standard error of fractal dimension
radius_worst	Worst or largest mean radius
texture_worst	Worst or largest mean texture
perimeter_worst	Worst or largest mean perimeter
area_worst	Worst or largest mean area
smoothness_worst	Worst or largest mean smoothness
compactness_worst	Worst or largest mean compactness
concavity_worst	Worst or largest mean concavity
concave points_worst	Worst or largest mean number of concave points
symmetry_worst	Worst or largest mean symmetry
fractal_dimension_worst	Worst or largest mean fractal dimension
target	Target variable: benign (0) or malignant (1)

Table 2: Features in the Breast Cancer Wisconsin Dataset

2 Exploratory Data Analysis (EDA) and Preprocessing

2.1 Feature summary

```
def feature_types(df):
  # Create an initial dataset with only the attributes
  var_df = pd.DataFrame(columns=['variable_name',
                                 'dtype',
                                'missing_percentage',
                                'flag', 'unique_values'])
  # Calculate the missing percentagesand sort them
  missing_percentages = df.isnull().mean() * 100
  missing_percentages = missing_percentages.sort_values(ascending=False)
  # For every column...
  for col in df.columns:
      variable_name = col
      dtype = df[col].dtype
      missing_percentage = missing_percentages[col]
      unique_values = df[col].nunique()
      if dtype=='object':
          flag = 'categorical'
      else:
          flag = 'numeric'
      #... save the name, dtype, missing perc, type and nunique
      var_df = pd.concat([var_df,
                          pd.DataFrame({'variable_name': [col],
                                        'dtype': [dtype],
                                        'missing_percentage': [missing_percentage],
                                        'flag': [flag],
                                        'unique_values': [unique_values]})],
                                        ignore_index=True)
  return var_df
```

Listing 1: This code shows some brief description of the data in order to acknowledge the type of preprocessing steps we're facing

variable_name	dtype	missing_percentage	flag	unique_values
radius_mean	float64	0.0	numeric	456
texture_mean	float64	0.0	numeric	479
perimeter_mean	float64	0.0	numeric	522
area_mean	float64	0.0	numeric	539
smoothness_mean	float64	0.0	numeric	474
compactness_mean	float64	0.0	numeric	537
concavity_mean	float64	0.0	numeric	537
concave points_mean	float64	0.0	numeric	542
symmetry_mean	float64	0.0	numeric	432
fractal_dimension_mean	float64	0.0	numeric	499
radius_se	float64	0.0	numeric	540
texture_se	float64	0.0	numeric	519
perimeter_se	float64	0.0	numeric	533
area_se	float64	0.0	numeric	528
smoothness_se	float64	0.0	numeric	547
compactness_se	float64	0.0	numeric	541
concavity_se	float64	0.0	numeric	533
concave points_se	float64	0.0	numeric	507
symmetry_se	float64	0.0	numeric	498
fractal_dimension_se	float64	0.0	numeric	545
radius_worst	float64	0.0	numeric	457
texture_worst	float64	0.0	numeric	511
perimeter_worst	float64	0.0	numeric	514
area_worst	float64	0.0	numeric	544
smoothness_worst	float64	0.0	numeric	411
compactness_worst	float64	0.0	numeric	529
concavity_worst	float64	0.0	numeric	539
concave points_worst	float64	0.0	numeric	492
symmetry_worst	float64	0.0	numeric	500
fractal_dimension_worst	float64	0.0	numeric	535
target	int64	0.0	numeric	2

Table 3: Brief description of the data

2.2 Correlation Analysis

The correlation between two variables refers to the statistical measure that captures the extent to which the variables change together.

It quantifies the strength and direction of the relationship between two sets of data. Correlation values range from -1 to 1, where:

- -1 indicates a perfect negative correlation: as one variable increases, the other decreases in a precisely linear manner.
- 0 indicates no correlation: changes in one variable do not predict changes in the other variable.
- 1 indicates a perfect positive correlation: as one variable increases, the other also increases in a precisely linear manner.

In order to remove features with high correlation between them I've implemented the next code snippet:

```
def correlation(df, threshold=0.9):
  var = list[df.columns]
  var_to_remove = []
  for i,j in zip(*np.where(np.abs(np.triu(df.corr()>threshold)))):
      if i!=j:
          if var[i] not in var_to_remove and var[j] not in var_to_remove:
              var_to_remove.append(var[i])
  df = df.drop(var_to_remove, axis=1)
  return df
```

Listing 2: Code snippet to remove attributes with high correlation

Feature Pair	Correlation Coefficient
radius_mean, perimeter_mean	0.9979
radius_mean, area_mean	0.9874
radius_mean, radius_worst	0.9695
radius_mean, perimeter_worst	0.9651
radius_mean, area_worst	0.9411
texture_mean, texture_worst	0.912
perimeter_mean, area_mean	0.9865
perimeter_mean, radius_worst	0.9695
perimeter_mean, perimeter_worst	0.9704
perimeter_mean, area_worst	0.9415
area_mean, radius_worst	0.9627
area_mean, perimeter_worst	0.9591
area_mean, area_worst	0.9592
concavity_mean, concave points_mean	0.9214
concave points_mean, concave points_worst	0.9102
radius_se, perimeter_se	0.9728
radius_se, area_se	0.9518
perimeter_se, area_se	0.9377
radius_worst, perimeter_worst	0.9937
radius_worst, area_worst	0.984
perimeter_worst, area_worst	0.9776

Table 4: Correlation Coefficients Between Features in the Breast Cancer Wisconsin Dataset

To visualize the changes in the dataset we can plot the correlation between features using a heatmap:

```
plt.figure(figsize=(15,10))
sns.heatmap(bcw_df.corr(), vmin=-1, vmax=1, cmap='coolwarm', annot=True)
plt.show()
```

Listing 3: Code snippet to visualize correlation heatmap

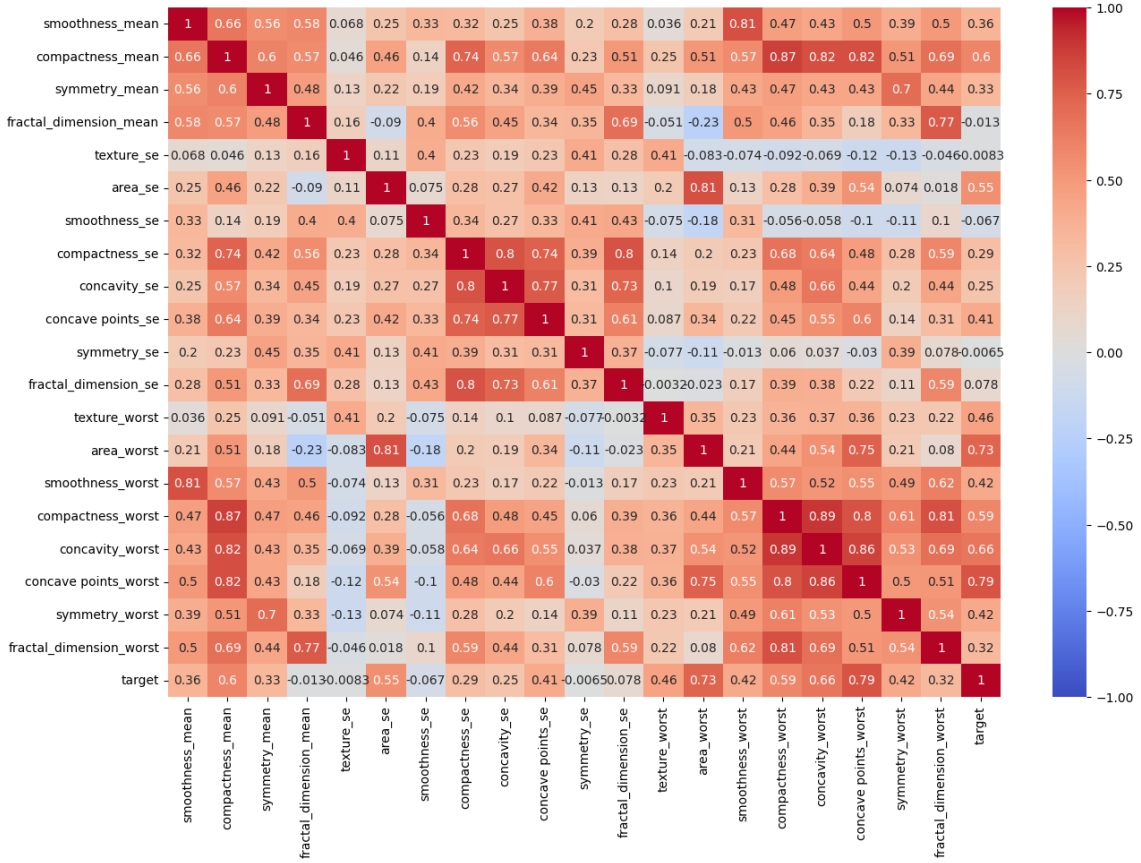

Figure 2: Correlation heatmap after cleaning

2.3 Normalization

Normalization refers to the process of scaling feature values to a range, typically so that the mean is 0 and the variance is 1.

It ensures that each feature contributes equally to the distance computation in machine learning algorithms.

The `StandardScaler` class standardizes features by subtracting the mean and then dividing by the standard deviation.

$$\mathcal{Z} = \frac{x - \mu}{\sigma} \tag{1}$$

```
plt.figure(figsize=(15,10))
sns.heatmap(bcw_df.corr(), vmin=-1, vmax=1, cmap='coolwarm', annot=True)
plt.show()
```

Listing 4: Code snippet to normalize the data using StandardScaler class from sklearn.preprocessing

smooth_mean	compactness_mean	symmetry_mean	fractal_dim_mean	target
1.6	3.3	2.2	2.3	1
-0.8	-0.5	0.0	-0.9	1
0.9	1.1	0.9	-0.4	1
3.3	3.4	2.9	4.9	1
0.3	0.5	-0.0	-0.6	1
2.2	1.2	1.0	1.9	1

Table 5: Peek at the post normalization dataset

2.4 Principal Component Analysis (PCA)

The main consideration when conducting PCA is determining the number of variables that can be reduced in our dataset without losing a significant percentage of explained variance.

When applying PCA to a dataset using sklearn in Python, the columns PC1, PC2, PC3, etc., represent the new dimensions resulting from dimensionality reduction. These new dimensions are called Principal Components (PC) and are calculated through a linear combination of the original features in the dataset.

The principal component PC1 has the highest variance and represents the main direction of maximum dispersion of the original data. PC2 represents the second main direction of maximum variance, and so on. As we move from PC1 to PC2, the variance decreases, representing a lesser amount of information compared to PC1.

Calculation of the PC's

1. **Computing the covariance matrix**. It shows how each variable changes in relation to others, indicating whether they tend to increase or decrease together. For each $X_1, X_2, ..., X_n$ features in the dataset (except the target class):

$$\sum = \begin{pmatrix} Var(X_1) & Cov(X_1, X_2) & \dots & Cov(X_1, X_n) \\ Cov(X_1, X_2) & Var(X_2) & \dots & Cov(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(X_n, X_1) & Cov(X_n, X_2) & \dots & Var(X_n) \end{pmatrix}$$

2. **Computing the Eigenvectors and Eigenvalues**. Eigenvalues are scalar values that signify the amount of variance captured by each eigenvector. Eigenvectors are vectors that represent directions in the data space.

$$\sum v = \lambda v \qquad \text{Where } v \text{ is the eigenvector and } \lambda \text{ is the eigenvalue}$$

3. **Selecting the PC's and project the eigenvectors**. First we choose the first k eigenvectors, where k is the number of dimensions you want to keep. These k eigenvectors form a matrix W that will be used to transform the original dataset.

   Then we project this matrix W into the data:

$$Y = W \times X$$

Visualizing PCA In order to visualize the PC's and what 'direction of maximum variance' means, I've coded a little example to visualize this concepts.

Upon selecting two scaled features from the dataset, namely `compactness_mean` and `smoothness_mean`, I proceeded to apply PCA setting $k = 2$, which implies $PC = 2$.

By visualizing the distribution of these two features alongside the two PCs derived from the PCA, one should observe that the directions of the PC vectors align with the dispersion trends of the data.

```
pca = PCA(n_components=2)
Xpca = pca.fit_transform(X.iloc[:, [0,1]])

plt.figure(figsize=(10, 8))
plt.scatter(bcw_norm_df.loc[bcw_norm_df.target == 1].iloc[:, 0],
            bcw_norm_df.loc[bcw_norm_df.target == 1].iloc[:, 1], color='orange')
plt.scatter(bcw_norm_df.loc[bcw_norm_df.target == 0].iloc[:, 0],
            bcw_norm_df.loc[bcw_norm_df.target == 0].iloc[:, 1], color='b')
plt.title('Visualization of Principal Components')
plt.legend(['Malignant', 'Benign'])
plt.arrow(0, 0, pca.components_[0,0], pca.components_[0,1],
          width=0.05, color='r')
plt.arrow(0, 0, pca.components_[1,0], pca.components_[1,1],
          width=0.05, color='r')
plt.annotate(r"$\bf{PC1}$",  pca.components_[0] + .1, fontsize=20,)
plt.annotate(r"$\bf{PC2}$",  pca.components_[1], fontsize=20,)
plt.xlabel('smoothness_mean')
plt.ylabel('compactness_mean')
plt.show()
```

Listing 5: Code snippet to visualize PC

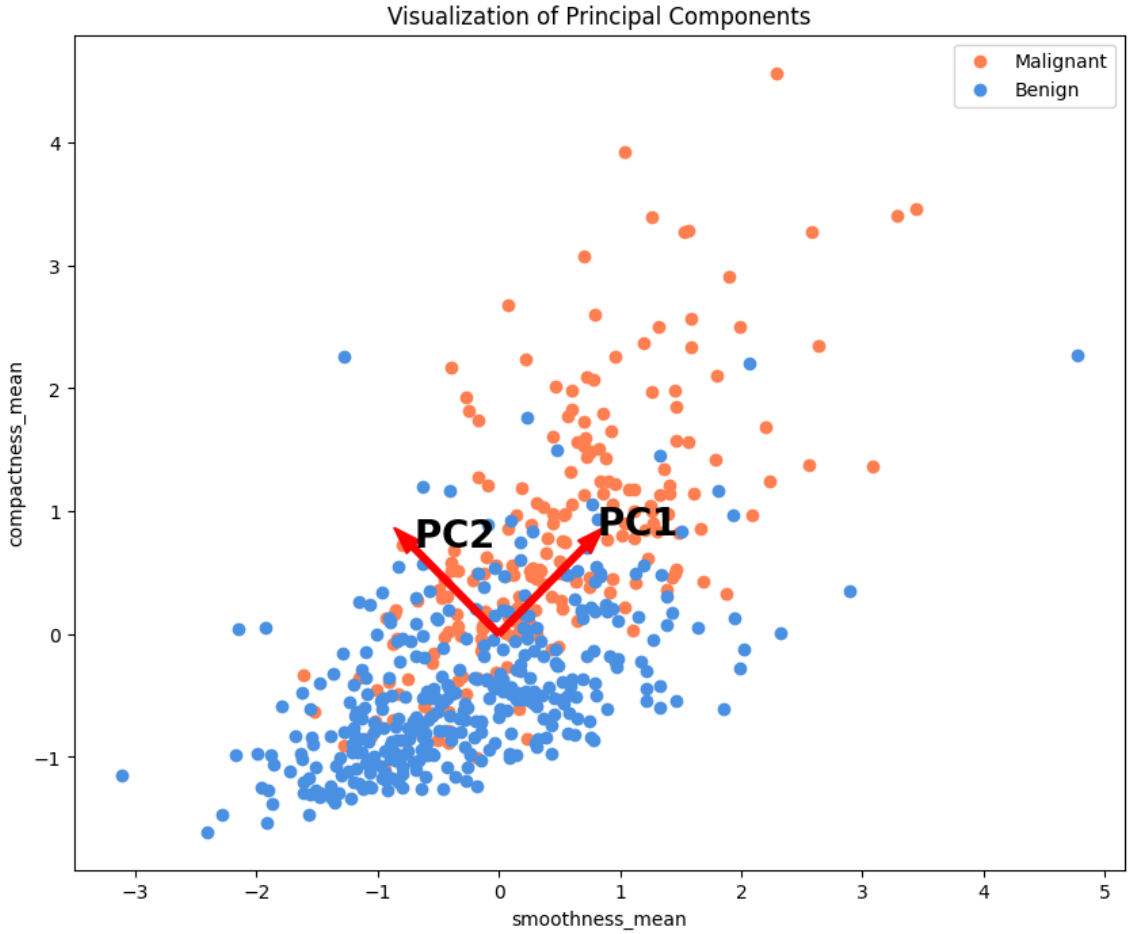

Figure 3: Visualization of the PCA output. PC1 and PC2 align with the directions of maximum dispersion of the data

PCA using Scikit-Learn Library When implementing PCA on the dataset we must take into account the fine-tuning process of the hyperparameter k. his parameter represents the count of eigenvectors that will constitute the transformation matrix W.

Each of these selected eigenvectors transforms into one Principal Component (PC) in the resultant dataset. Thus, the true hyperparameter of interest is actually the quantity of Principal Components.

The goal is to select a suitable number of PCs that capture a significant proportion of the explained variance inherent in the original dataset.

```
from sklearn.decomposition import PCA

X, y = bcw_norm_df.drop('target', axis=1), bcw_norm_df['target']
pca = PCA()
pca.fit(X)
explained_var = np.cumsum(np.round(pca.explained_variance_ratio_,
                                   decimals=2) * 100)

plt.figure(figsize=(10,8))
plt.axhline(100, linestyle='--', color='gray')
plt.plot(explained_var, color='#4A90E2', label='Explained Variance')
points = plt.scatter(range(len(explained_var)), explained_var,
                     color='coral', label='Data Points')
for i, txt in enumerate(explained_var):
    plt.annotate(f"{txt:.1f}", (i, txt), textcoords="offset points",
                 xytext=(0, 5), ha='center')
plt.xlabel('Num. of Principal Components')
plt.xticks(np.arange(0, 25, 5))
plt.yticks(np.arange(40, 110, 10))
plt.ylabel('Explained variance (%)')
plt.xticks(range(0, 20, 5))
plt.legend()
plt.show()
```

Listing 6: Code snippet to compute the explained variance for each Principal Component and visualize it

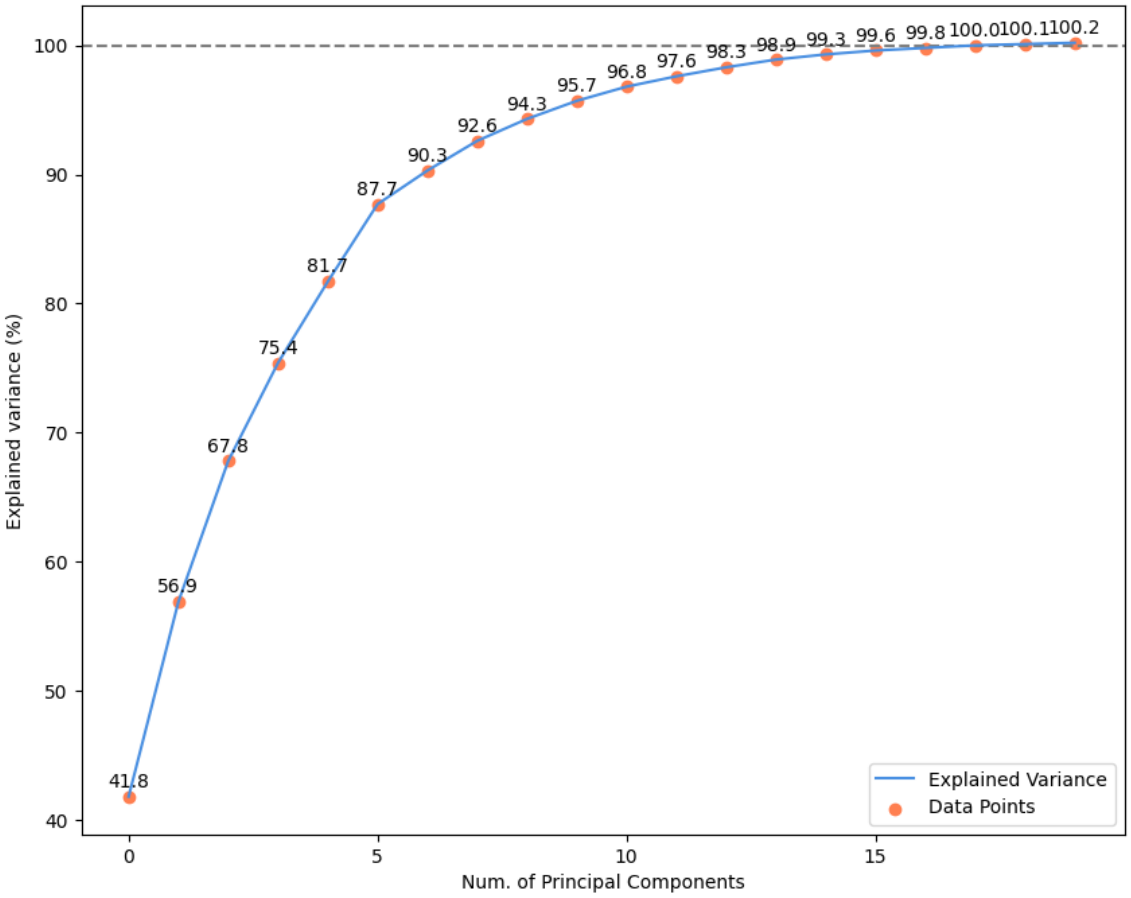

Figure 4: Explained variance (%) versus number of Principal Components

Upon observation of the plot, I've determined that using $PC = 10$ preserves 95.7% of the original variance, offering a notable reduction in dataset attributes.

```
Xpca = pca.transform(X)[:, :10]
bcw_pca_df = pd.DataFrame(Xpca, columns=[f'PC {i+1}' for i in range(10)])
bcw_pca_df['target'] = y
```

Listing 7: Code snippet to apply PCA to the BCW dataset (previously normalized)

2.5 Data Pipeline

In scikit-learn, a Pipeline is essentially a sequence of data processing elements, where each element is a tuple containing a name and an instance of a transformer or estimator. The elements are executed in a sequence, with data flowing from one step to the next.

This data structure makes it easier to apply preprocessing steps after having previously studied the dataset.

```
from sklearn.pipeline import Pipeline

bcw_df = correlation(bcw_df)
pipeline = Pipeline([
          ('Normalization', StandardScaler()),
          ('PCA', PCA(n_components=10))])

X, y = bcw_df.drop(['target'], axis=1), bcw_df['target']

bcw_pipeline_df = pipeline.fit_transform(X)
bcw_preproc_df = pd.DataFrame(bcw_pipeline_df,
                 columns=[f'PC {i+1}' for i in range(10)])
bcw_preproc_df['target'] = y
path = os.path.join('/content/drive/MyDrive/Datasets/SI_final_project',
                    'bcw_preproc.csv')
bcw_preproc_df.to_csv(path, index=False)
```

Listing 8: Code snippet to apply PCA to the BCW dataset (previously normalized)

PC 1	PC 2	PC 3	PC 4	(...)	target
7.4	-1.5	-1.1	-0.6	(...)	1
-0.6	-2.2	0.9	0.4	(...)	1
3.3	-1.9	0.6	-0.4	(...)	1
11.4	2.5	-5.5	-1.5	(...)	1
1.0	-0.3	1.6	0.2	(...)	1

Table 6: Peek at the final dataset

3 Workflow

To accurately assess various machine learning models developed with Scikit-Learn, we need to outline the workflow for each model. This includes determining how to train the model with the data and how to measure its accuracy on test data.

3.1 Cross Validation

To evaluate how well a supervised machine learning model performs, we typically divide our data into a *training* set and a *test* set. If we're fine-tuning the model, we might also create a *validation* set.

Given the random nature of these splits, there's a significant chance of *overfitting* the training data.

A solution to this problem would be using *Cross Validation* (CV). The test set would still be held out for final evaluation, and the validation set is no longer needed when doing CV.

In the most basic approach, called k-fold CV, the training set is split into k smaller sets, called folds. For each split of the data, one fold is selected as validation set, and the other k-1 folds are selected as training set. This way, for each split of the data the validation set changes and the final performance is evaluated by computing the mean of each performance measure for all the folds.

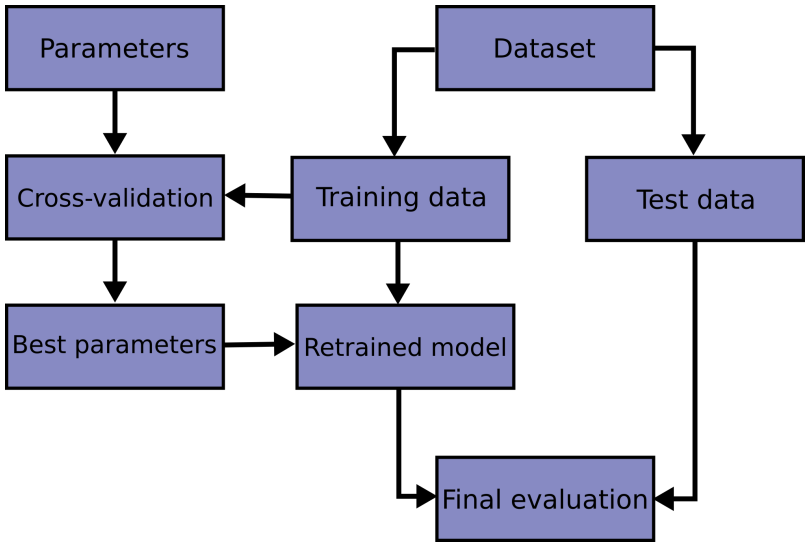

Figure 5: Workflow for this document[2]

3.2 Assessing the problem of unbalanced target class

K-fold cross validation is a great technique to evaluate the performance of a ML model for different training data, however, upon exploration of the distribution of target class samples, we can conclude that there is an imbalance among the data (see Image 6).

```
class_count = bcw_df['target'].value_counts()
plt.bar(class_count.index, class_count.values,
        color=['#4A90E2', 'coral'], label='Unbalanced Target Class')
plt.xlabel('Classes')
plt.ylabel('Count')
plt.title('Target Class Balance')
plt.xticks(class_count.index, [str(i) for i in class_count.index])
plt.show()
```

Listing 9: Code snippet to visualize the target class imbalance

Figure 6: Target class distribution of samples

To address this problem, I'll perform CV using the **Stratified K-Fold Cross Validation (SCV)**. This CV procedure is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

3.3 Performance evaluation

This section covers the different metrics and plots that will be used to evaluate the performance of each ML model implemented in this project.

- **Confusion matrix**. The confusion matrix will be plotted for each model using a matplotlib or seaborn heatmap.
- **Traditional measures**. Accuracy, Precision, Recall and F1-Score.
- **Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC)**. When appropriate, the ROC curve will be used to evaluate the model performance for each threshold.
- **Other curves**. For models that dont use a threshold value for the classification of their samples (such as Depth for Decision Tree), other curves will be plotted in order to fine tune this kind of parameters.
- **Final evaluation**. The metrics of each model will be saved into a `.csv` file. This data sheet will then be analyzed in order to choose the best model with the best hyperparameters for this classification problem.

The dataset will be first split into Train and Test. Then, I'll apply SCV for the Train set, obtaining Train and Validation for each fold. This data will be used to fine-tune the model, finding the best values for the hyperparameters. The Test data will be used to get the final performance metrics of the tuned model.

```
from sklearn.model_selection import train_test_split

X, y = bcw_preproc_df.drop(['target'], axis=1), bcw_df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.33,
                                                    random_state=42)

bcw_test = pd.DataFrame(columns=X_test.columns, data=X_test)
bcw_test['target'] = y_test

bcw_train = pd.DataFrame(columns=X_train.columns, data=X_train)
bcw_train['target'] = y_train

path = os.path.join('/content/drive/MyDrive/Datasets/SI_final_project',
                    'bcw_train.csv')
bcw_train.to_csv(path, index=False)
path = os.path.join('/content/drive/MyDrive/Datasets/SI_final_project',
                    'bcw_test.csv')
bcw_test.to_csv(path, index=False)
```

Listing 10: Code snippet to save the test and train `.csv` files

3.4 Decision Boundaries

Visualizing the decision boundaries of a machine learning model is a good practice for several reasons.

It provides an intuitive understanding of how the algorithm partitions the feature space to make classifications. This is particularly helpful in cases where interpretability and understandability are critical. By plotting the decision boundaries, one can grasp how changes in feature values could potentially affect the classification outcome.

It also helps in identifying areas where the model might be too rigid or too flexible, indicating underfitting or overfitting.

While the model itself may be trained on multiple features, the decision_boundary function focuses only on two features for the purpose of visualization. The function creates a grid of points in the 2D space defined by these two selected features and then makes predictions using the trained model for those points. Since in the PC space PC1 and PC2 represent the maximum direction of variance on our data, I'll provide some useful visualization.

However, using just two principal components attempts to set a balance between simplicity and informativeness, allowing for a more manageable and insightful visual representation of the model's decision-making process.

In conclusion, if the model structure allows us, I'll be plotting the decision boundary for the reduced PC1-PC2 dataset using the code 11.

```python
def decision_boudary(model, min_vals, max_vals, num_features, feature_ids=[0,1]):
    """
    - Arguments:
      model: Previously trained ML model.
      min_vals: 2D array or list with the minimum values of the regular grid.
      max_vals: 2D array or list with the maximum values of the regular grid.
      num_features: Total number of input variables for the model
                    (even if only 2 are represented, the model may have many more).
      feature_ids: Indices of the variables that we are going to use.
    """
    # First we create a grid that covers the search space in
    # the square defined by min_vals and max_vals
    ranges = [np.arange(min_v-.1, max_v+.1, 0.05)
              for min_v, max_v in zip(min_vals, max_vals)]
    xx, yy = np.meshgrid(*ranges)

    # Since we're only focusing on PC1 and PC2, we fill the rest with zero
    X = np.zeros([xx.ravel().shape[0], num_features])
    X[:, feature_ids[0]] = xx.ravel()
    X[:, feature_ids[1]] = yy.ravel()

    # We evaluate each point in the grid with the previously trained model.
    # This way, we'll create a reasonable aproximation of the decision that
    # our model would make for each point in the PC space.
    zz = model.predict(X).reshape(xx.shape)
    return xx, yy, zz
```

Listing 11: Function to plot the decision boundary (surface) for any model. I took inspiration from [3]

```
pcs = [1, 2]

# Decision boundary
xx, yy, zz = decision_boudary(INPUT_MODEL, X_train.min(0)[pcs],
                              X_train.max(0)[pcs],
                              X_train.shape[1], feature_ids=pcs)

# Plotting
cmap = colors.ListedColormap(['lightcoral', 'mediumseagreen'])
plt.figure(figsize=(8,8))
plt.pcolormesh(xx, yy, zz.astype(float), cmap=cmap)
clrs = [ ('lightcoral', 'mediumseagreen')[int(y_i)] for y_i in y_train]
plt.scatter(X_train.values[:, pcs[0]],
            X_train.values[:, pcs[1]], ec='k', color=clrs, s=25)
plt.xlabel(f'PC{pcs[0]}')
plt.ylabel(f'PC{pcs[1]}')
plt.show()
```

Listing 12: Train and Test split for the reduced dataset (only PC1 and PC2)

4 Naive Bayes Classifier

4.1 Model Training

4.1.1 Training and Validation

```
# Import ML model
from sklearn.naive_bayes import GaussianNB

scv = StratifiedKFold(n_splits=10, random_state=True, shuffle=True)
X, y = bcw_train.drop('target', axis=1), bcw_train['target']

fig, ax = plt.subplots(figsize=(8,6))
for i, (train, validation) in enumerate(scv.split(X, y)):
  gnb = GaussianNB()
  gnb.fit(X.iloc[train], y.iloc[train])
  # Note that train and validation are indexes among X and y !!
  RocCurveDisplay.from_estimator(gnb, X.iloc[validation],
                                 y.iloc[validation],
                                 name="ROC fold {}".format(i),
                                 alpha=0.3, lw=1, ax=ax)

  y_pred = gnb.predict(X.iloc[validation])
  # We save the metrics
  metrics_df.loc[f'Fold {i}', 'Accuracy'] = accuracy_score(y.iloc[validation],
                                                           y_pred)
  metrics_df.loc[f'Fold {i}', 'Precision'] = precision_score(y.iloc[validation],
                                                             y_pred)
  metrics_df.loc[f'Fold {i}', 'Recall'] = recall_score(y.iloc[validation],
                                                       y_pred)
  metrics_df.loc[f'Fold {i}', 'F1-Score'] = f1_score(y.iloc[validation],
                                                     y_pred)


ax.plot(np.linspace(0, 1, 100), np.linspace(0, 1, 100), 'r--', label='y=x')
plt.title('ROC curve of Gaussian Naive Bayes')
plt.show()
```

Listing 13: Training of Gaussian Naive Bayes

4.1.2 Performance in training stage and fine-tuning hyperparameters

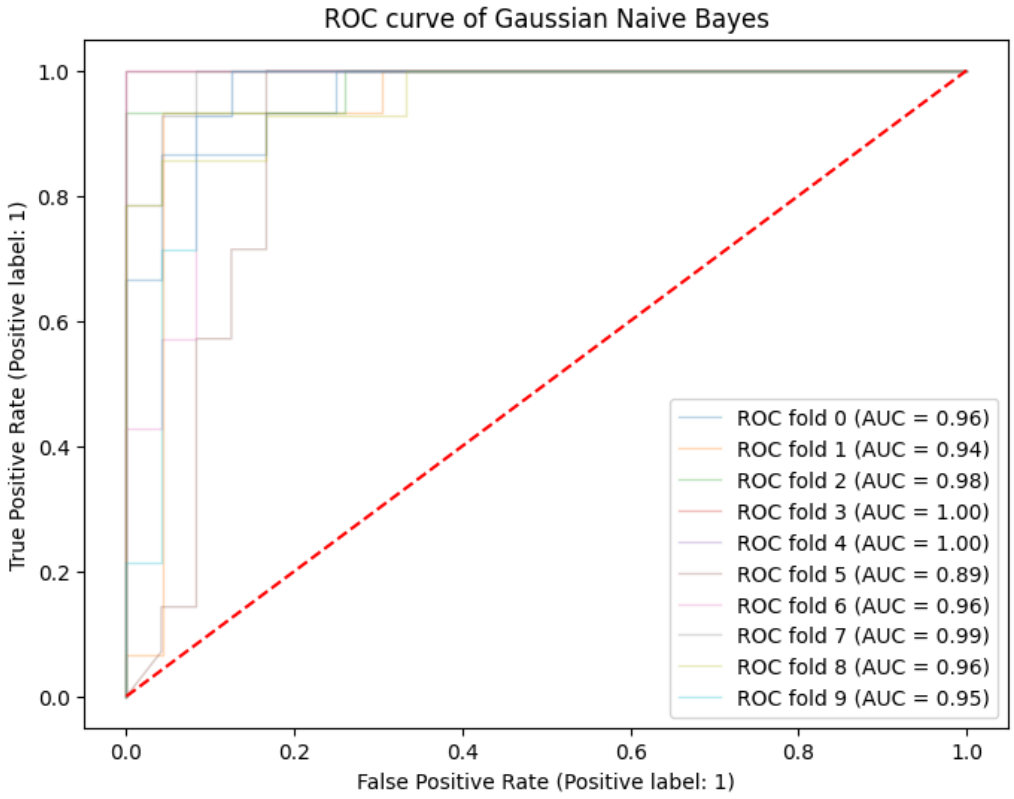

Figure 7: ROC and AUC for each fold

	Accuracy	Precision	Recall	F1-Score
Fold 0	0.8	0.9	0.7	0.8
Fold 1	0.9	0.9	0.8	0.9
Fold 2	0.9	0.9	0.9	0.9
Fold 3	1.0	1.0	0.9	1.0
Fold 4	0.9	1.0	0.7	0.8
Fold 5	0.8	0.8	0.6	0.7
Fold 6	0.9	0.8	0.8	0.8
Fold 7	0.9	1.0	0.8	0.9
Fold 8	0.9	1.0	0.8	0.9
Fold 9	0.9	0.9	0.9	0.9

Table 7: Performance metrics for each fold

	Accuracy	Precision	Recall	F1-Score
max	1.0	1.0	0.9	1.0
min	0.8	0.8	0.6	0.7
mean	0.9	0.9	0.8	0.8
std	0.1	0.1	0.1	0.1

Table 8: Caption

From the ROC curve, it is evident that some classifiers, specifically from folds 3 and 4, achieved perfection with an AUC value of 1.0. Other folds also performed near-optimally. This observation aligns with the performance metrics provided for each fold in table 7.

In table 8, a summary of the training performance metrics is presented. A consistent Standard Deviation (std) is observed across all folds for each performance metric. While most metrics exhibit similar values, the Recall stands out as the lowest, registering a minimum value of 0.6.

Finally, upon observation of the ROC curve, we can estimate that the best **threshold** value is around 0.4.

4.2 Testing

```
X, y = bcw_test.drop('target', axis=1), bcw_test['target']
y_pred = (gnb.predict_proba(X)[:, 1] >= 0.4).astype(int)
final_metrics = pd.DataFrame(data=
                             {'Accuracy': accuracy_score(y, y_pred),
                              'Precision': accuracy_score(y, y_pred),
                              'Recall': recall_score(y, y_pred),
                              'F1-Score': f1_score(y, y_pred)}, index=[0])
# Compute the confusion matrix
cm = confusion_matrix(y, y_pred)
# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Listing 14: Testing of Gausian Naive Bayes

4.2.1 Final performance evaluation

Accuracy	Precision	Recall	F1-Score
0.88	0.88	0.82	0.83

Table 9: Final performance metrics

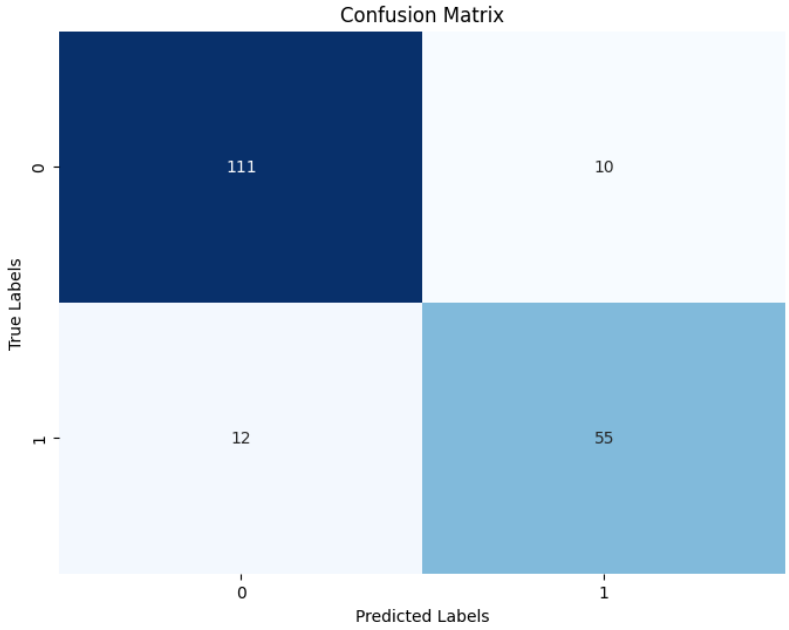

Figure 8: Confusion matrix for GNB

The GNB classifier's performance, as visualized through the confusion matrix, suggests a robust predictive capability. Out of the samples, the model accurately predicted 111 instances as class 0 and 55 instances as class 1. However, it misclassified 10 instances of class 0 as class 1 and 12 instances of class 1 as class 0. This indicates a tendency of the model to produce false positives.

Looking at the performance metrics below the matrix, the model achieved an accuracy of 88.3%, suggesting that it correctly predicted the majority of the instances. The precision score of 88.3% indicates that of all the predicted positive instances, 88.3% were indeed positive. The recall, at 82.1%, denotes that out of all the actual positive instances, the model was able to correctly identify 82.1% of them. Finally, the F1-score stands at 83.3%, reflecting a balanced trade-off between precision and recall.

Overall, the GNB classifier demonstrates a commendable performance, but there's room for improvement, especially in minimizing false positives.

5 K-Nearest Neighbours Classifier

5.1 Model Training

5.1.1 Training and Validation

The code on this one particular model is quite extensive, to provide a more comfortable reading I've divided it into sections.

```
# Import CV related libraries
from sklearn.model_selection import StratifiedKFold
# Import performance related libraries
from sklearn.metrics import (accuracy_score, recall_score, precision_score,
                             f1_score, confusion_matrix, RocCurveDisplay)

metrics_df = pd.DataFrame(columns=['Accuracy',
                                   'Precision',
                                   'Recall',
                                   'F1-Score'],
                          index=[f"Fold {i}" for i in np.arange(0, 10)])

from sklearn.neighbors import KNeighborsClassifier

scv = StratifiedKFold(n_splits=10, random_state=42, shuffle=True)
X, y = bcw_train.drop('target', axis=1), bcw_train['target']

# List to store the mean of each fold
average_metrics = []
avg_f1_values = []
avg_accuracy_values = []
```

Listing 15: Libraries and variables

```
fig, ax = plt.subplots(figsize=(12, 8))
for k in range(2, 21):
    fold_metrics = {'F1-Score': [], 'Accuracy': []}
    for i, (train, validation) in enumerate(scv.split(X, y)):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X.iloc[train], y.iloc[train])

        y_pred = knn.predict(X.iloc[validation])
        fold_metrics['F1-Score'].append(
                                    f1_score(y.iloc[validation], y_pred))
        fold_metrics['Accuracy'].append(
                                    accuracy_score(y.iloc[validation], y_pred))
        metrics_df.loc[f'Fold {i}', 'Accuracy'] =
                accuracy_score(y.iloc[validation], y_pred)
        metrics_df.loc[f'Fold {i}', 'Precision'] =
                precision_score(y.iloc[validation], y_pred)
        metrics_df.loc[f'Fold {i}', 'Recall'] =
                recall_score(y.iloc[validation], y_pred)
        metrics_df.loc[f'Fold {i}', 'F1-Score'] =
                f1_score(y.iloc[validation],  y_pred)

    # Calculate average metrics for this k value
    avg_f1 = np.mean(fold_metrics['F1-Score'])
    avg_accuracy = np.mean(fold_metrics['Accuracy'])
    # These two lists save the data to plot the line
    avg_f1_values.append(avg_f1)
    avg_accuracy_values.append(avg_accuracy)
    ax.scatter(k, avg_f1, color='b', marker='x')
    ax.scatter(k, avg_accuracy, color='r', marker='x')
    # Save the neighbours and the fold score for that neightbour
    average_metrics.append((k, avg_f1, avg_accuracy))
```

Listing 16: Fitting and saving useful data

```
ax.plot(range(2, 21), avg_f1_values, color='b', alpha=0.3)
ax.plot(range(2, 21), avg_accuracy_values, color='r', alpha=0.3)

ax.set_xlabel('Number of Neighbors (k)')
ax.set_ylabel('Score')
ax.set_title('F1-Score and Accuracy vs Number of Neighbors')
ax.legend(['F1-Score', 'Accuracy'])
plt.show()

# Find the best k value based on average F1-Score
best_k = max(average_metrics, key=lambda x: x[1])[0]
# Output the best k value and first few average metrics for review
best_k, average_metrics[:5]
```

Listing 17: Plotting the results to find the best K value

5.1.2 Performance in training stage and fine-tuning hyperparameters

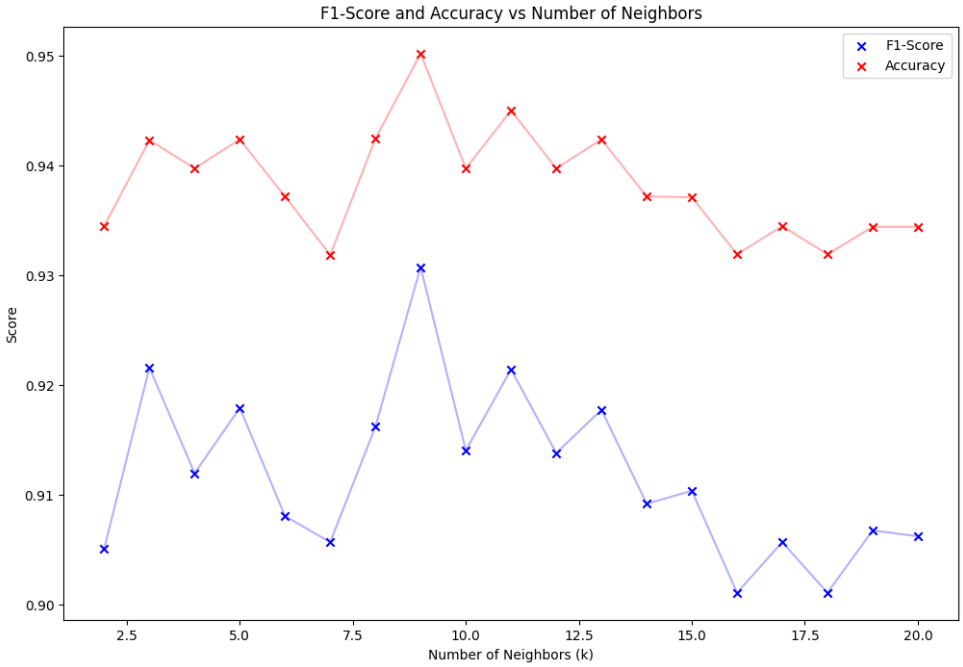

Figure 9: Best K value plot for Accuracy and F1-Score

Upon observation of the plot 9 we may approximate that the best k value is in $(7.5, 10)$ range. If we show the variable `best_k`, we can conclude that the best k value is 9. This evidence can be backed with the data shown in table 10

This k value will be used to compute the final performance metrics of the model on the test data.

	Neighbour	F1-Score	Accuracy
0	2	0.905	0.934
1	3	0.922	0.942
2	4	0.912	0.940
3	5	0.918	0.942
4	6	0.908	0.937
5	7	0.906	0.932
6	8	0.916	0.942
7	9	0.931	0.950
8	10	0.914	0.940
9	11	0.921	0.945
10	12	0.914	0.940
11	13	0.918	0.942
12	14	0.909	0.937
13	15	0.910	0.937
14	16	0.901	0.932
15	17	0.906	0.934
16	18	0.901	0.932
17	19	0.907	0.934
18	20	0.906	0.934

Table 10: Mean of the performance metrics of each fold for every neighbour

5.2 Testing

```
# Initialize the KNN classifier and fit
X, y = bcw_train.drop('target', axis=1), bcw_train['target']
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X, y)
X_test, y_test = bcw_test.drop('target', axis=1), bcw_test['target']

y_pred = knn_best.predict(X_test)

final_metrics = pd.DataFrame(data=
                             {'Accuracy': accuracy_score(y, y_pred),
                              'Precision': accuracy_score(y, y_pred),
                              'Recall': recall_score(y, y_pred),
                              'F1-Score': f1_score(y, y_pred)}, index=[0])
final_metrics['Model'] = 'KNN'
# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Listing 18: Testing of K Nearest Neighbours

5.2.1 Final performance evaluation

	Accuracy	Precision	Recall	F1-Score
0	0.936	0.936	0.881	0.908

Table 11: Final performance metrics

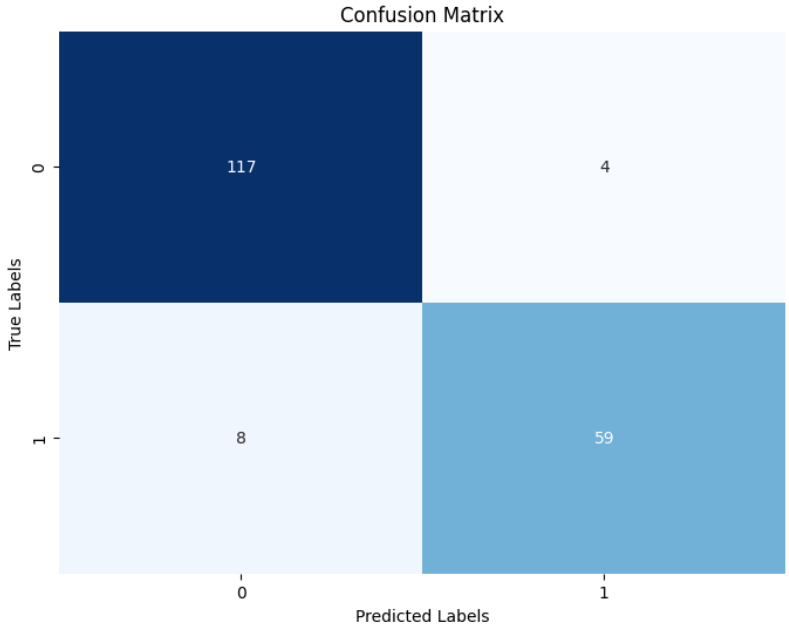

Figure 10: Confusion matrix for GNB

The K-Nearest Neighbors (KNN) classifier showcased in the provided confusion matrix demonstrates strong predictive capabilities. The model accurately predicted 117 instances as class 0 and 59 as class 1. On the flip side, it made a few errors, misclassifying 4 instances of class 0 as class 1 and 8 instances of class 1 as class 0. This highlights a minor inclination of the model to produce false negatives over false positives. Delving into the aggregated performance metrics, the model attained an accuracy of approximately 93.6%, illustrating its aptitude in making correct predictions for the vast majority of the samples.

The precision score, at 93.6%, suggests that out of all the instances predicted as positive, 93.6% were genuinely positive. Meanwhile, the recall of 88.1% indicates that the model identified 88.1% of all actual positive samples correctly. The F1-score, which harmonizes precision and recall, stands at 90.8%, signifying a well-balanced model in terms of its ability to minimize both false positives and false negatives.

In conclusion, the KNN classifier demonstrates an impressive performance with only slight room for further optimization.

5.3 Understanding how k-NN Classifiers work: Decision Boundaries

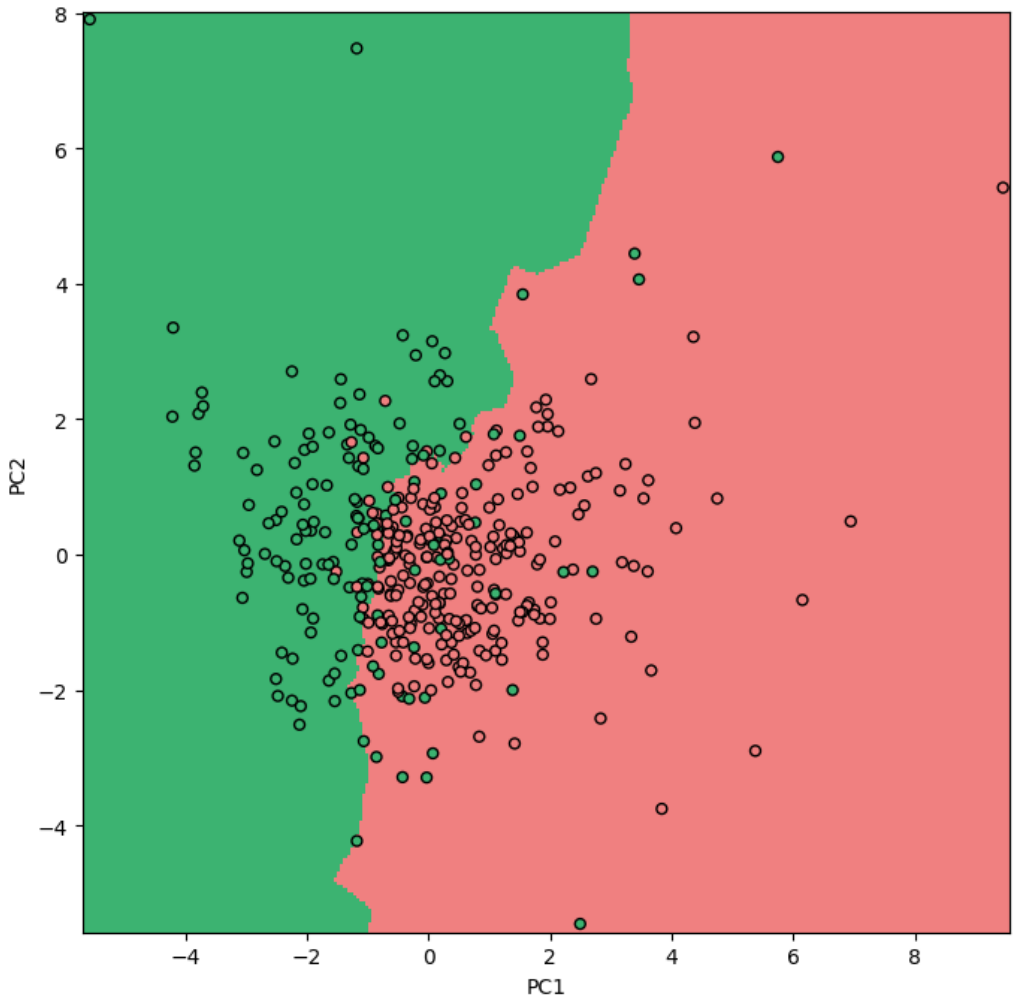

Figure 11: Decision Boundary for k-NN Classifier with k=8

The decision boundary plot for the KNN classifier11 reveals interesting insights into the model's performance on the training set. While the classifier appears to separate the two classes reasonably well in the PC feature space, there are notable exceptions.

Some yellow points (representing one class) are situated within the red decision region designated for the other class. These misclassified points suggest that the model is not perfectly capturing the underlying data distribution. Such misclassifications indicate areas where the model may benefit from further tuning or from the inclusion of additional features to better discriminate between the classes.

6 Decision Tree Classifier

6.1 Model Training

In the initial stage of our decision tree modeling process, we systematically explore three distinct criteria for determining the quality of a split within the tree: "gini"[4], "entropy"[5], and "log_loss"[6]. The Gini impurity is gauged using the "gini" criterion, while both the "entropy" and "log_loss" criteria evaluate the quality of splits based on the Shannon information gain.

As we proceed with each of these criteria, the model undergoes training using a Stratified 10-fold cross-validation approach. This training process is conducted for a range of tree depths from 3 to 40.

Subsequent to the training, we embark on a comprehensive analysis wherein we evaluate the outcomes of each criterion based on two critical metrics: depth-to-accuracy and depth-to-F1 score ratios. By delving into these comparisons, our objective is to discern the criterion that offers optimal generalization. The ideal criterion would be one that, with minimal tree depth, yields the pinnacle of both accuracy and F1 score. By prioritizing these factors, we ensure that our final model is both efficient and highly effective in its predictions.

6.1.1 Training and Validation

```
# Import CV related libraries
from sklearn.model_selection import StratifiedKFold
# Import performance related libraries
from sklearn.metrics import (accuracy_score, recall_score, precision_score,
                             f1_score, confusion_matrix, RocCurveDisplay)

from sklearn.tree import DecisionTreeClassifier

scv = StratifiedKFold(n_splits=10, random_state=42, shuffle=True)
X, y = bcw_train.drop('target', axis=1), bcw_train['target']
criteria = ['gini', 'entropy', 'log_loss']
criteria_metrics = pd.DataFrame(columns=criteria, index=['Accuracy', 'F1-Score'])
```

Listing 19: Libraries and variables

```
for criterion in criteria:
    fig, ax = plt.subplots(figsize=(12, 8))

    average_metrics = []  # Reset for each criterion
    avg_f1_values = []
    avg_accuracy_values = []

    for depth in np.arange(3, 40):
        fold_metrics = {'F1-Score': [], 'Accuracy': []}

        for i, (train, validation) in enumerate(scv.split(X, y)):
            dtree = DecisionTreeClassifier(criterion=criterion, max_depth=depth)
            dtree.fit(X.iloc[train], y.iloc[train])

            y_pred = dtree.predict(X.iloc[validation])
            fold_metrics['F1-Score'].append(f1_score(y.iloc[validation],
                                                     y_pred))
            fold_metrics['Accuracy'].append(accuracy_score(y.iloc[validation],
                                                     y_pred))

        # Calculate average metrics for this k value
        avg_f1 = np.mean(fold_metrics['F1-Score'])
        avg_accuracy = np.mean(fold_metrics['Accuracy'])
        # These two lists save the data to plot the line
        avg_f1_values.append(avg_f1)
        avg_accuracy_values.append(avg_accuracy)
        # Plotting the average F1-Score and Accuracy for this k value
        ax.scatter(depth, avg_f1, color='b', marker='x')
        ax.scatter(depth, avg_accuracy, color='r', marker='x')
        # Save the neighbours and the fold score for that neightbour
        average_metrics.append((depth, avg_f1, avg_accuracy))
    (...)
```

Listing 20: Training using SCV and saving important performance data

```
(...) # Inside for criterion in criteria for loop (!)
ax.plot(range(3, 40), avg_f1_values, color='b', alpha=0.3)
ax.plot(range(3, 40), avg_accuracy_values, color='r', alpha=0.3)

# Find the maximum values and their corresponding depth values
max_f1 = max(avg_f1_values)
depth_max_f1 = avg_f1_values.index(max_f1) + 3
max_accuracy = max(avg_accuracy_values)
depth_max_accuracy = avg_accuracy_values.index(max_accuracy) + 3
criteria_metrics[criterion] = [(max_accuracy.round(3), depth_max_accuracy),
                               (max_f1.round(3), depth_max_f1)]
#Plot
ax.set_xlabel('Decision tree depth')
ax.set_ylabel('Score')
ax.set_title(f'F1-Score and Accuracy vs Depth using {criterion} criterion')
ax.legend(['F1-Score', 'Accuracy'])
plt.show()
```

Listing 21: Training using SCV and saving important performance data

6.1.2 Performance in training stage and fine-tuning hyperparameters

The executed code finishes with the generation of three distinct graphs, each corresponding to a specific criterion employed during the decision tree evaluation process.

Each graph prominently displays two functions: The first function delineates the relationship between the decision tree depth and the accuracy achieved at each depth level. The second function, in parallel, plots the F1 score against the same depth spectrum.

Here I present this three plots:

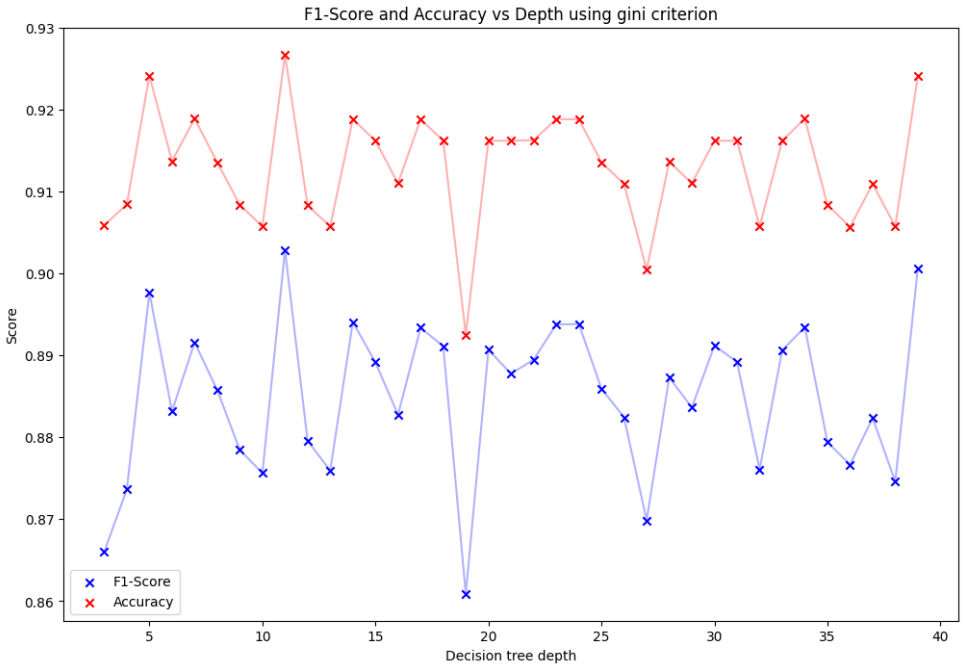

Figure 12: Accuracy, F1-Score vs depth for Gini criterion

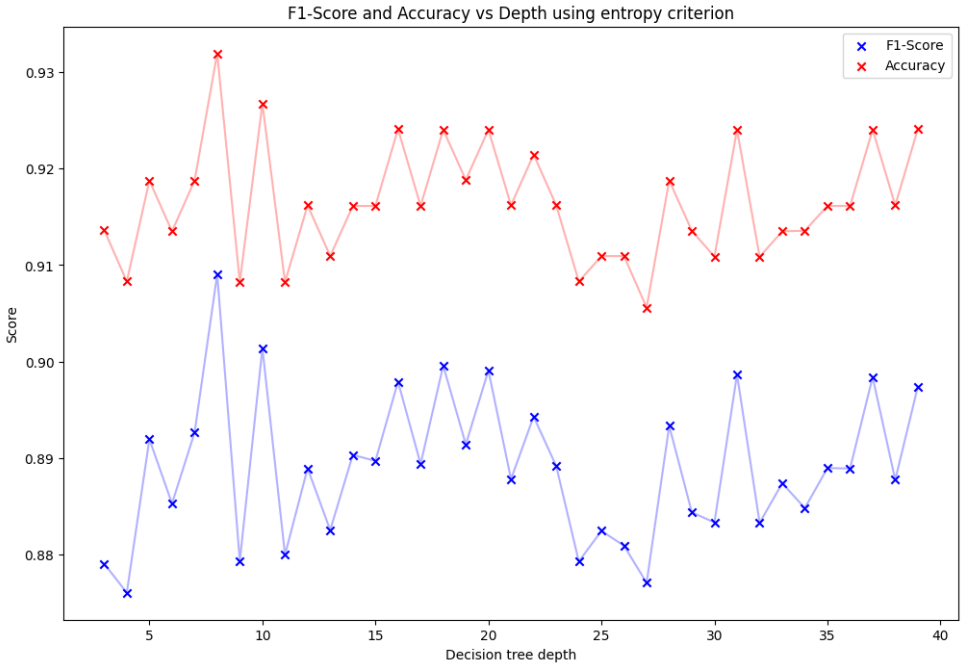

Figure 13: Accuracy, F1-Score vs depth for Entropy criterion

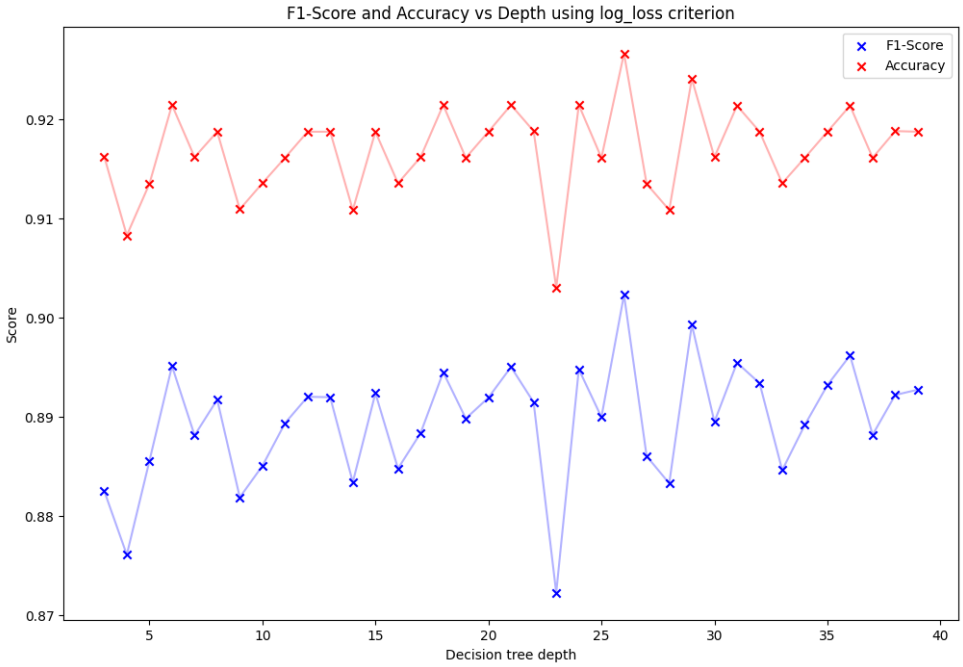

Figure 14: Accuracy, F1-Score vs depth for Logloss criterion

Finally, for each criterion I've chosen to represent the best accuracy and f1-score with their respective depth:

	gini	entropy	log_loss
Accuracy	(0.927, 11)	(0.932, 8)	(0.927, 26)
F1-Score	(0.903, 11)	(0.909, 8)	(0.902, 26)

Table 12: Accuracy and F1 Score for Decision Treee criteria for measuring the quality of a split. The data is presented as: (performance, depth). This data was obtained from plots 121314

- For the "gini" criterion: The highest accuracy achieved is 0.927, at a tree depth of 11. The corresponding F1-score at this depth is 0.903.
- For the "entropy" criterion: The peak accuracy is slightly higher at 0.932, achieved at a lesser tree depth of 8. The F1-score at this depth is 0.909, also showing a slight improvement over the "gini" criterion.
- For the "log_loss" criterion: The accuracy peaks at 0.927, similar to the "gini" criterion. However, this is achieved at a much deeper tree depth of 26. The F1-score for this depth is 0.902, which is marginally lower than the "gini" criterion and notably lower than the "entropy" criterion.

In summary, the "entropy" criterion provides the best performance in terms of both accuracy and F1-score at a relatively shallow tree depth of 8. While the "gini" and "log_loss" criteria achieve similar accuracy levels, "gini" does so at a much shallower depth than "log_loss". This suggests that, for the BCW dataset, using the "entropy" criterion with depth = 8 is be the most efficient choice for model generalization and computational efficiency.

6.2 Testing

```
# Initialize the Decision tree classifier and fit
X_train, y_train = bcw_train.drop('target', axis=1), bcw_train['target']

dtree_best = DecisionTreeClassifier(criterion='entropy', max_depth=8)
dtree_best.fit(X_train, y_train)

X_test, y_test = bcw_test.drop('target', axis=1), bcw_test['target']
y_pred = dtree_best.predict(X_test)

final_metrics = pd.DataFrame(data=
                             {'Accuracy': accuracy_score(y, y_pred),
                              'Precision': accuracy_score(y, y_pred),
                              'Recall': recall_score(y, y_pred),
                              'F1-Score': f1_score(y, y_pred)}, index=[0])
final_metrics['Model'] = 'DTREE'
# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Listing 22: Testing stage for depth=8 and criterion: 'entropy'

6.2.1 Final performance Evaluation

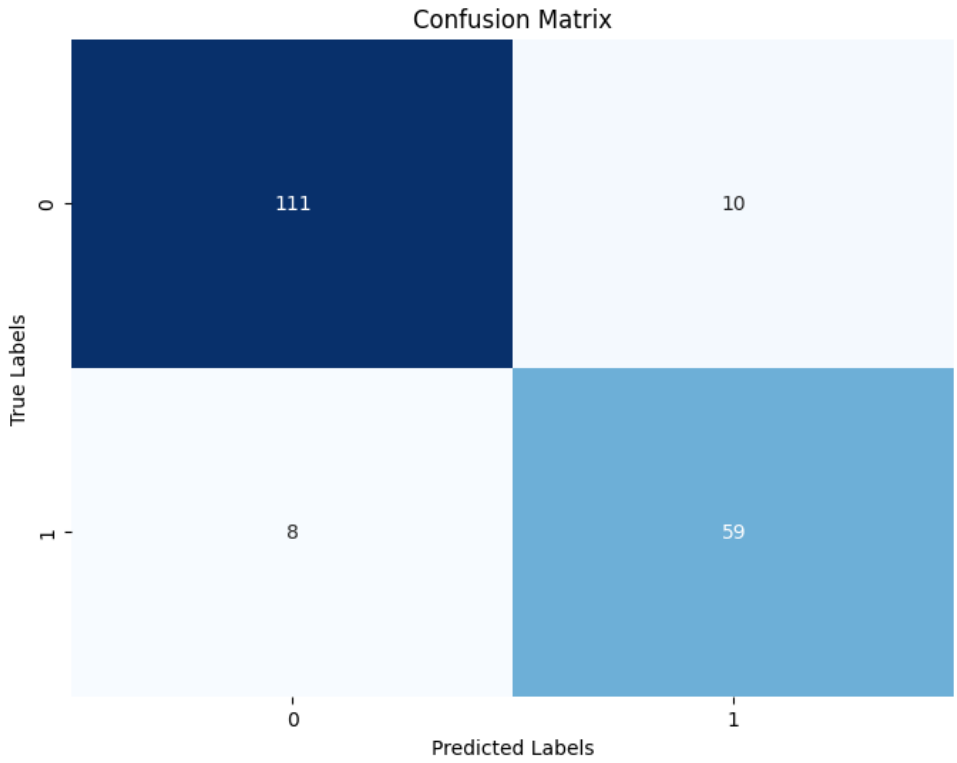

Figure 15: Confusion Matrix for Decision Tree Model

The confusion matrix15 shows that out of all the instances predicted as class '0', 111 were correctly classified (true negatives), while 10 were misclassified (false positives). Similarly, for class '1', 59 instances were correctly classified (true positives), and 8 were wrongly predicted as class '0' (false negatives).

	Accuracy	Precision	Recall	F1-Score
0	0.904	0.904	0.881	0.868

Table 13: Performance measures for Decision Tree

The table 13 summarizes the effectiveness of the classifier. An accuracy of 0.904 indicates that the model correctly predicted approximately 90.4% of the instances in the test set. Precision, at 0.904, tells us that 90.4% of the instances predicted as class '0' were actually class '0'. A recall of 0.881 suggests that out of all the actual class '0' instances, 88.1% were correctly identified by the model. The F1-Score is 0.868, suggesting a balanced performance between precision and recall.

6.2.2 Plotting the tree

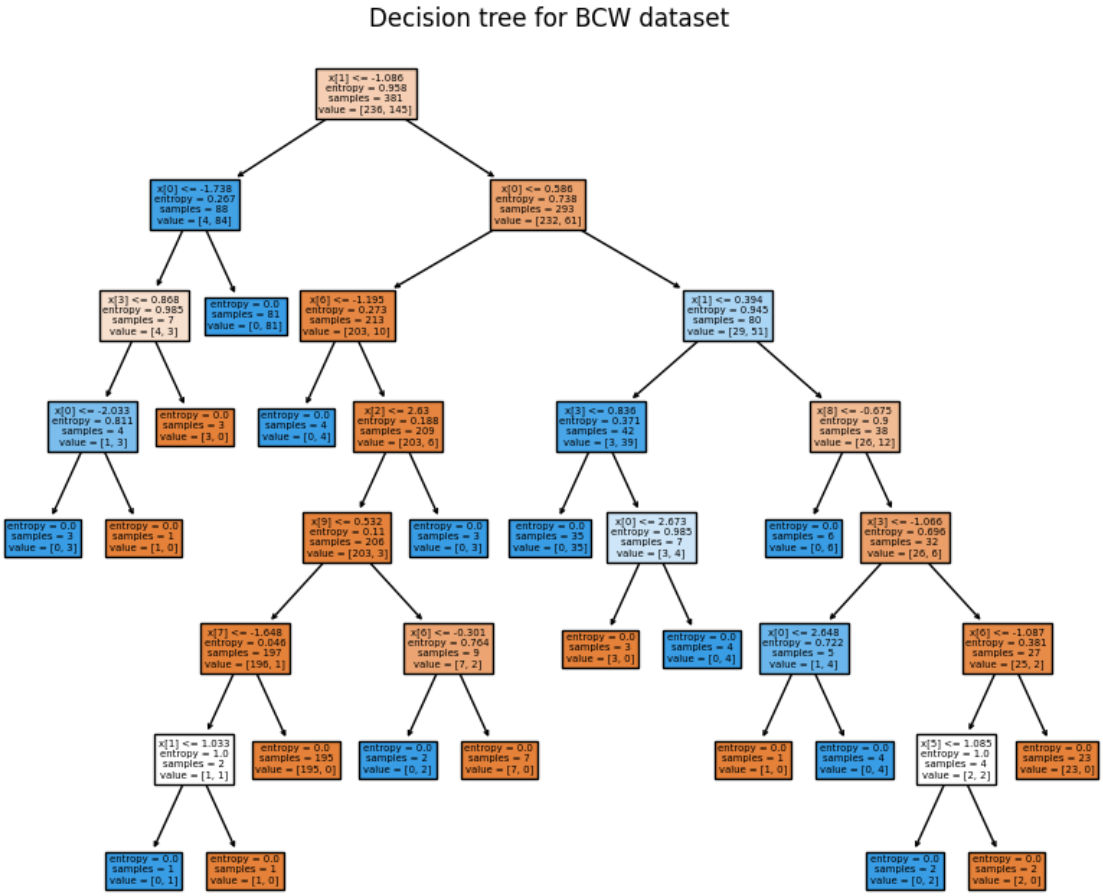

Figure 16: Decision Tree

This is the graphical representation of the series of decisions made by the classifier to segregate the data based on certain conditions.

Each node in the tree represents a condition or a question about one of the features, leading to branches and further questions until a decision about the class label is made. The visualized tree starts by examining the feature labeled "X[2]" and making decisions based on its value, splitting the data further as we move down the tree.

The colors in the nodes represent the majority class in that specific subset of data, and the deeper you go into the tree, the more specific the decisions become.

The tree was plotted using `from sklearn.tree import plot_tree` library and `plot_tree(dtree_best, filled=True)` method.

6.3 Understanding how Decision Trees work: Decision Boundaries

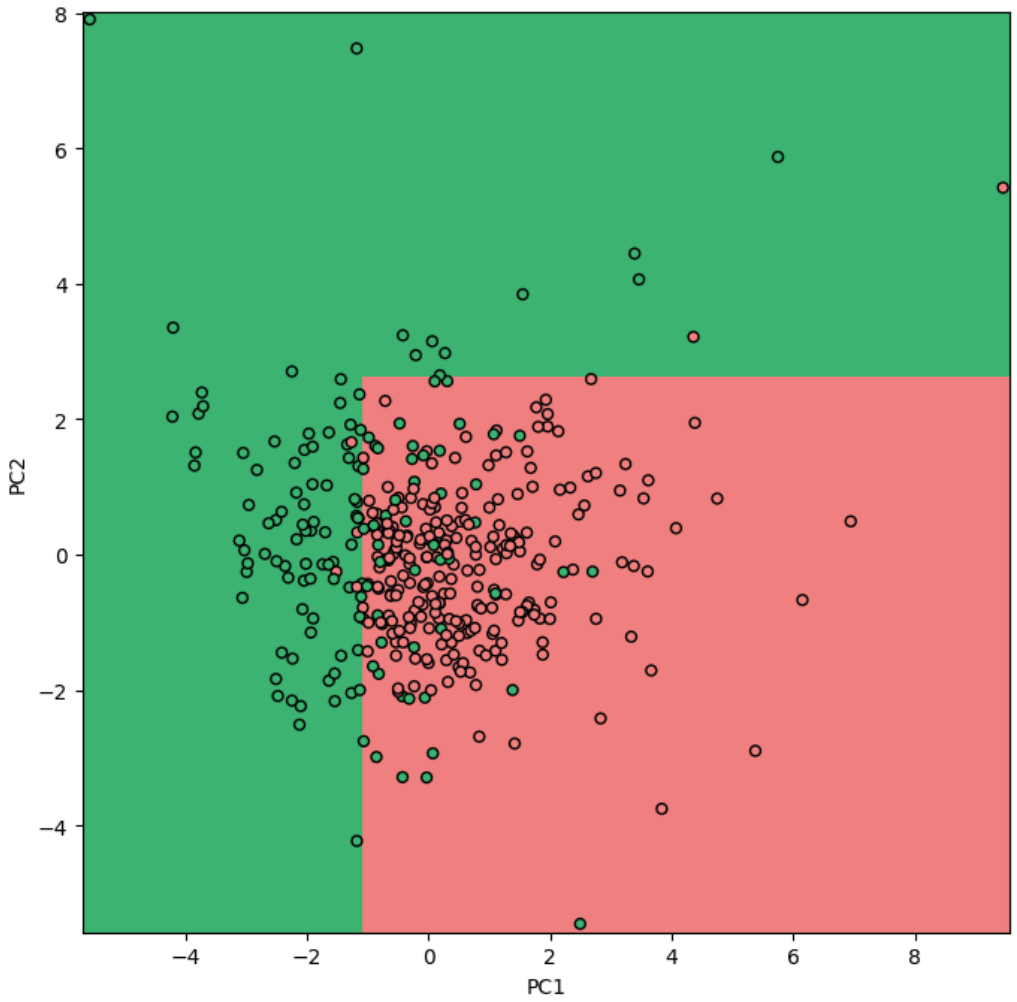

Figure 17: Decision Boundary for PC1 and PC2, a maximum depth of 8 and 'entropy' as the decision criterion

In the plot, the decision boundaries demarcate the regions of the feature space corresponding to different class labels. These boundaries are the result of the splits made by the decision tree during the training process.

The decision tree algorithm looks for the best way to partition the feature space to segregate classes optimally, based on the criteria set (entropy). The boundaries are essentially a visualization of these optimal partitions.

In this plot, any new data point falling within a particular shaded area would be classified as belonging to the class that the shaded area represents.

7 Support Vector Machine (SVM)

7.1 Algorithm

SVM are a class of supervised learning algorithms used for classification and regression tasks. The core idea is to find a hyperplane that best separates the data into different classes while maximizing the margin between them.

The margin is defined as the distance between the closest points (support vectors) of different classes to the hyperplane.

SVM allows for linear as well as non-linear classification by employing various types of kernel functions, including linear, polynomial, radial basis function (RBF), and sigmoid and other hyperparameters like the regularization parameter C and γ.

Hyperparameters

- **Kernel**. The kernel is a function that computes the similarity between two data points in a higher-dimensional space without explicitly transforming the data. It allows the algorithm to perform non-linear classification by implicitly mapping the input features into a higher-dimensional space where the data can be linearly separable.

  The kernels that will be explored in this document include linear, polynomial (default, so , radial basis function (RBF), and sigmoid.

$$\text{Linear Kernel: } K(x, y) = x \cdot y \tag{2}$$

$$\text{Polynomial Kernel: } K(x, y) = (x \cdot y + c)^d \tag{3}$$

$$\text{RBF (Radial Basis Function) Kernel: } K(x, y) = e^{-\gamma ||x-y||^2} \tag{4}$$

$$\text{Sigmoid Kernel: } K(x, y) = \tanh(\alpha x \cdot y + c) \tag{5}$$

- C **value**. The C value, or regularization parameter, controls the trade-off between maximizing the margin and minimizing the classification error.

  A smaller C value prioritizes a larger margin at the cost of allowing some misclassifications, resulting in a smoother decision boundary. A really small C could lead to underfitting.

  A larger C value aims to minimize misclassification, potentially at the expense of a more complex, wiggly decision boundary. A big enough C could lead to overfitting.

- γ **value**. The γ value is a hyperparameter specific to certain kernel types like the RBF kernel. It controls the shape and reach of the decision boundary around individual data points.

  A small γ value results in a more flexible, wiggly decision boundary, capturing more data points in its vicinity.

  A large γ value leads to a tighter, more isolated decision boundary, focusing on points close to the separating hyperplane.

  For this specific hyperparameter the fine tuning wont be so exhaustive. The sklearn library has a default value of $\gamma = \frac{1}{\text{n features} \times \text{X.var()}}$, which can be indicated with `gamma='scale'`.

The principal hyperparameters subject to optimization in this study are the kernel type and the regularization parameter C.

For each candidate C value, both Accuracy and F1-Score metrics will be evaluated across four distinct kernels. The C value that achieves the highest performance in terms of both Accuracy and F1-Score will be selected for further analysis. Subsequent comparisons, including an examination of decision boundaries, will be conducted across the four kernel types.

If the linear kernel emerges as the optimal choice, there wont be any further optimization with respect to the γ value.

7.2 Model Training

7.2.1 Training and Validation

```
# Import CV related libraries
from sklearn.model_selection import StratifiedKFold
# Import performance related libraries
from sklearn.metrics import (accuracy_score, recall_score, precision_score,
                             f1_score, confusion_matrix, RocCurveDisplay)
# Import ML model
from sklearn.svm import SVC

scv = StratifiedKFold(n_splits=10, random_state=42, shuffle=True)
X, y = bcw_train.drop('target', axis=1), bcw_train['target']
kernels = ['linear', 'rbf', 'poly', 'sigmoid']
C_values = [1e-3, 1e-2, 0.1, 1, 10, 100]

# Let's initialize an empty DataFrame to hold the results
all_results = pd.DataFrame([], columns=['Performance', 'Kernel', 'C', 'PType'])
```

Listing 23: Libraries, variables and fine-tuning values

```
for c_val in C_values:
    for ker in kernels:
        fold_metrics = {'F1-Score': [], 'Accuracy': []}

        for i, (train, validation) in enumerate(scv.split(X, y)):
            svc_aux = SVC(C=c_val, kernel=ker)
            svc_aux.fit(X.iloc[train], y.iloc[train])

            y_pred = svc_aux.predict(X.iloc[validation])
            fold_metrics['F1-Score'].append(f1_score(y.iloc[validation],
                                                     y_pred))
            fold_metrics['Accuracy'].append(accuracy_score(y.iloc[validation],
                                                     y_pred))

        # Fill the DataFrame for Accuracy
        aux_df = pd.DataFrame({
            'Performance': [np.mean(fold_metrics['Accuracy'])],
            'Kernel': [ker],
            'C': [c_val],
            'PType': ['Accuracy']
        })
        # Append Accuracy DataFrame to the overall results
        all_results = pd.concat([all_results, aux_df], ignore_index=True)
        # Fill the DataFrame for F1-Score
        aux_df = pd.DataFrame({
            'Performance': [np.mean(fold_metrics['F1-Score'])],
            'Kernel': [ker],
            'C': [c_val],
            'PType': ['F1-Score']
        })
        # Append F1-Score DataFrame to the overall results
        all_results = pd.concat([all_results, aux_df], ignore_index=True)
    # Plotting for each C value
    plt.figure(figsize=(10, 6))
    sns.pointplot(x="Kernel", y="Performance", hue="PType",
                  data=all_results[all_results['C'] == c_val],
                  palette="Set2", linestyles="-", join=True)
    plt.title(f'C={c_val}')
    plt.ylim(0, 1)
    plt.show()
```

Listing 24: Training using SCV and plotting the Performance measure for every Kernel for each C value (Accuracy and F1-Score (hue))

7.2.2 Performance in training stage and fine-tuning hyperparameters

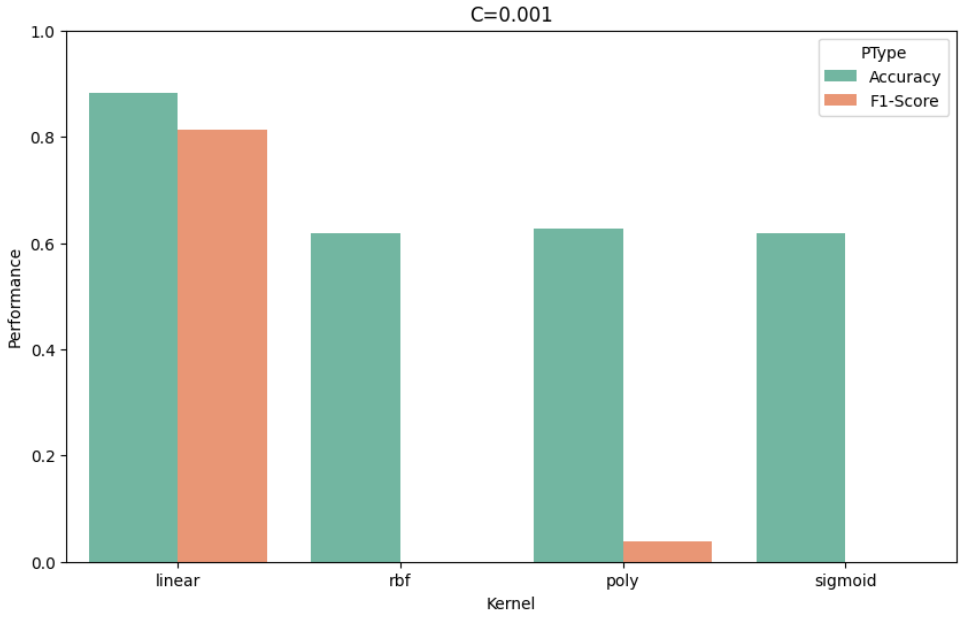

Figure 18: Acurracy and F1-Score performance measures for each Kernel for $C = 0.001$

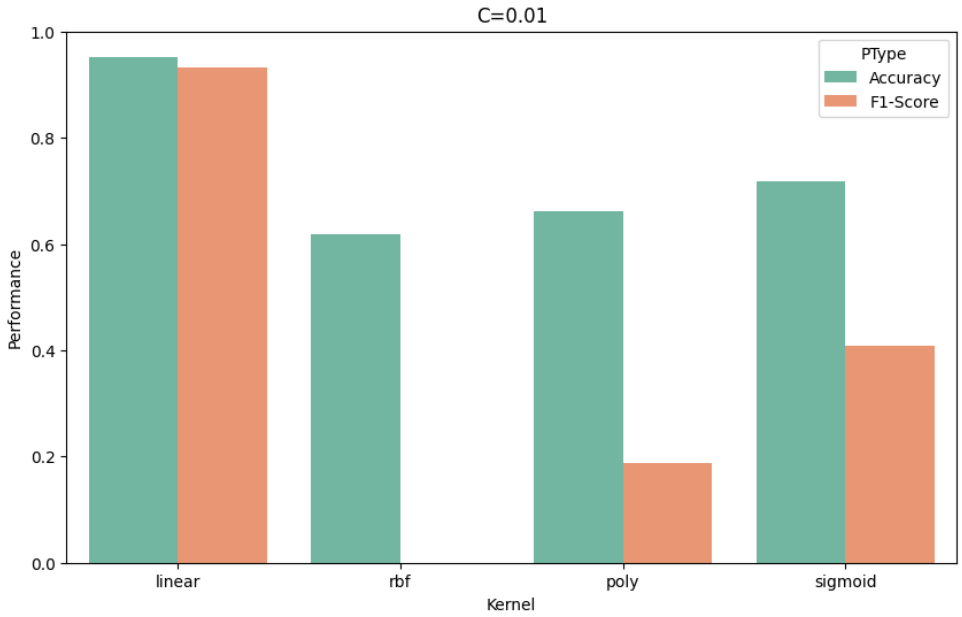

Figure 19: Acurracy and F1-Score performance measures for each Kernel for $C = 0.01$

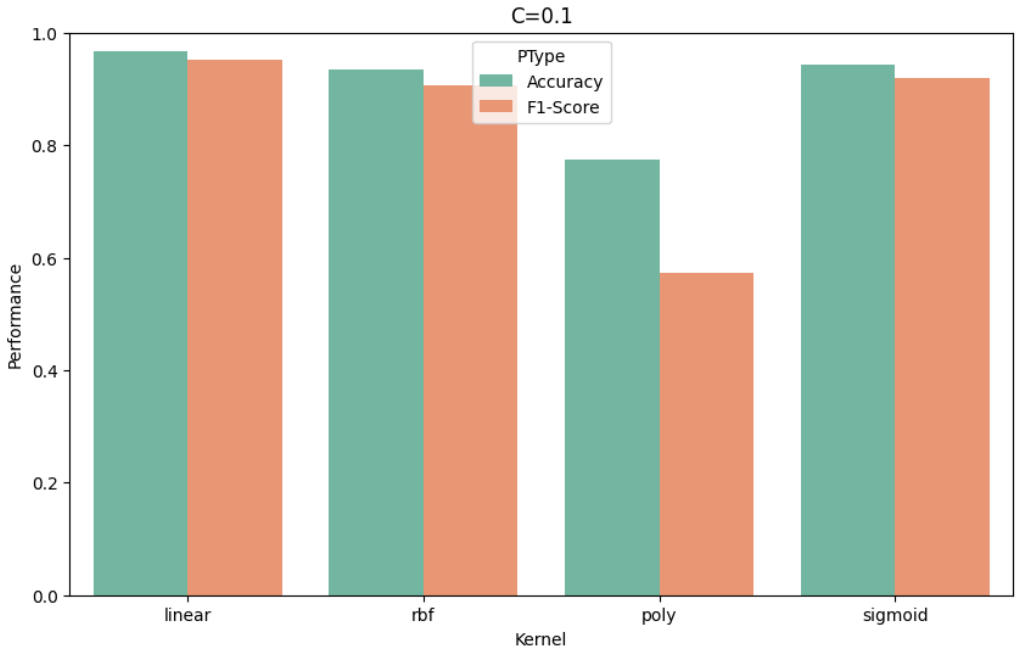

Figure 20: Acurracy and F1-Score performance measures for each Kernel for $C = 0.1$

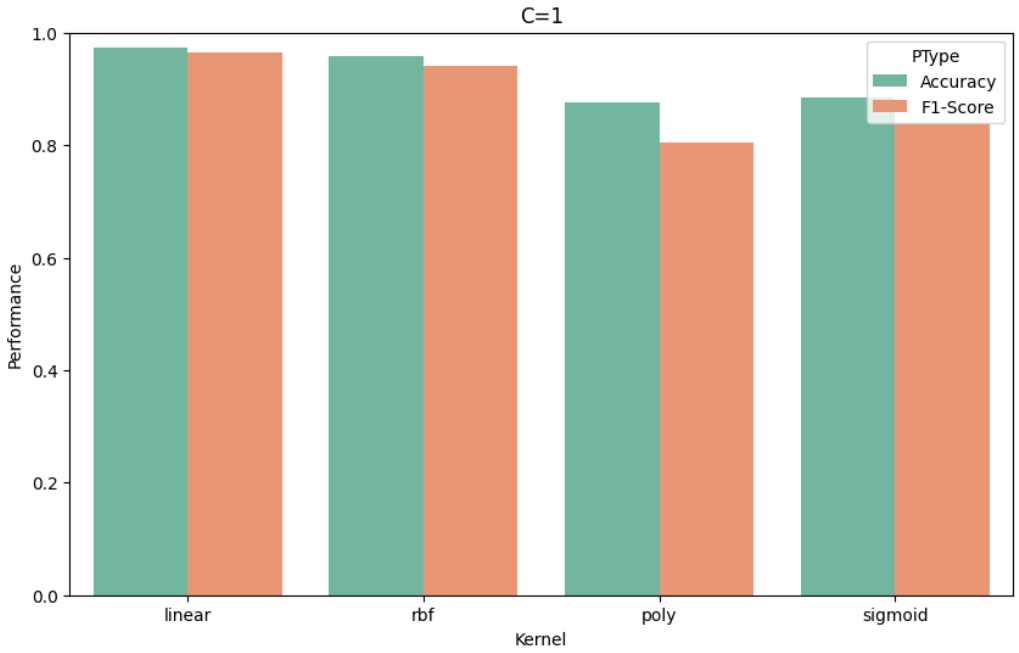

Figure 21: Acurracy and F1-Score performance measures for each Kernel for $C = 1$

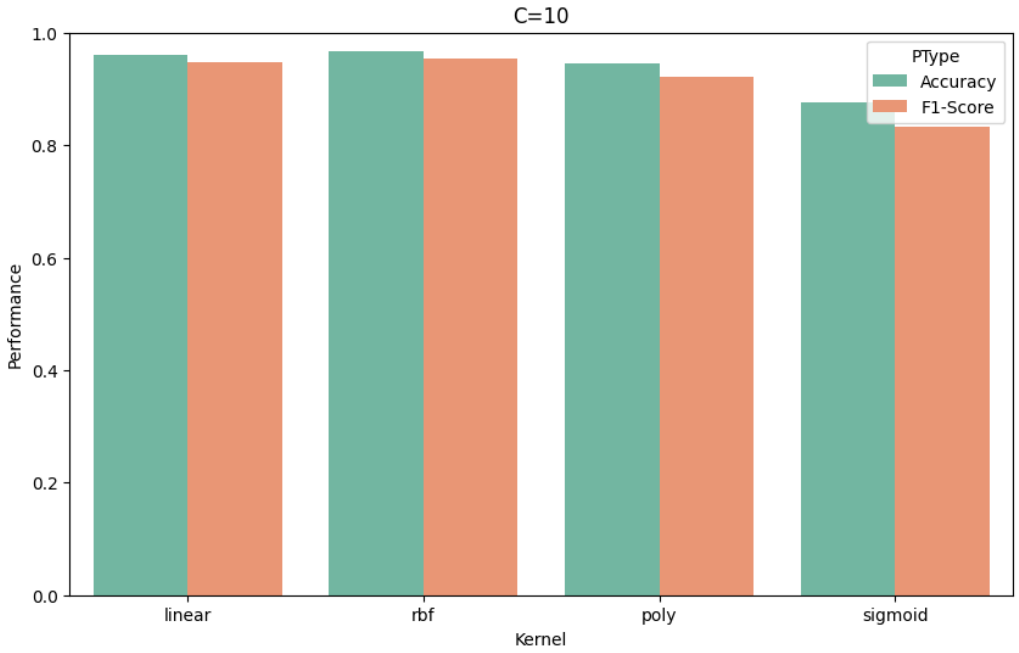

Figure 22: Acurracy and F1-Score performance measures for each Kernel for $C = 10$

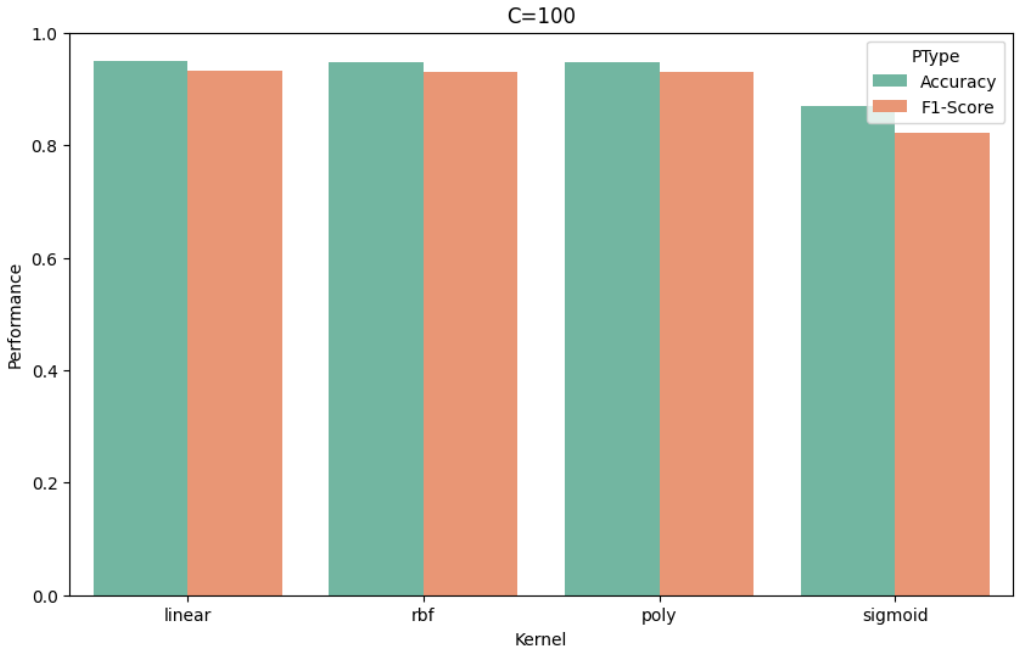

Figure 23: Acurracy and F1-Score performance measures for each Kernel for $C = 100$

Based on the data plotted, I have created the tables 14 and 15, which summarize the maximum performance measures of the training stage for each kernel:

	Kernel	C	Performance	PType
24	linear	1.000	0.974	Accuracy
44	poly	100.000	0.947	Accuracy
34	rbf	10.000	0.966	Accuracy
22	sigmoid	0.100	0.942	Accuracy

Table 14: Accuracy values corresponding to the chosen C values for each Kernel

	Kernel	C	Performance	PType
25	linear	1.000	0.965	F1-Score
45	poly	100.000	0.929	F1-Score
35	rbf	10.000	0.953	F1-Score
23	sigmoid	0.100	0.919	F1-Score

Table 15: F1-Score values corresponding to the chosen C values for each Kernel

The 'linear' kernel with a C value of 1.000 appears to be the best option for both Accuracy and F1-Score. Specifically, it achieves the highest Accuracy of 0.974 and an F1-Score of 0.965, outperforming the other kernels on both metrics. These high scores on both performance measures suggest that this combination of kernel and C value might generalize well to test data, offering a balanced trade-off between precision and recall, as well as a low misclassification rate.

The 'rbf' kernel with a C value of 10.000 also performs well, particularly in terms of Accuracy (0.966) and F1-Score (0.953). However, it slightly trails the 'linear' kernel on both measures. Given that the 'linear' kernel performs better and is computationally less complex than the 'rbf' kernel, I will proceed with the 'linear' kernel and a C value of 1.000 for further evaluations and tests, even though the rbf kernel might provide a more interesting approach due to its non-linear nature.

7.2.3 How the Kernel and C value affect the decision boundary

In this section I have decided to study how the interplay between the kernel type and the regularization parameter C influences the decision boundary in SVM.

Given that the choice of kernel and C value critically governs the model's capacity to segregate classes, understanding their impact on the decision boundary serves as a key step in model selection and fine-tuning. Through a series of visualizations, I aim to provide an intuitive grasp of how different combinations of these hyperparameters shape the decision landscape, and consequently, the model's performance on both training and test data.

```
for c_val in C_values:
    for ker in kernels:
        fold_metrics = {'F1-Score': [], 'Accuracy': []}
        for i, (train, validation) in enumerate(scv.split(X_train, y_train)):
            svc = SVC(C=c_val, kernel=ker)
            svc.fit(X_train.iloc[train], y_train.iloc[train])
            y_pred = svc.predict(X_train.iloc[validation])
            fold_metrics['F1-Score'].append(f1_score(y_train.iloc[validation],
                                                     y_pred))
            fold_metrics['Accuracy'].append(accuracy_score(y_train.iloc[validation],
                                                     y_pred))
        # Plotting decision boundary
        ax = axes.flat[plot_idx]  # Get the current axis
        # We calculate the decision boundary
        xx, yy, zz = decision_boudary(svc, X_train.min(0)[pcs],
                                      X_train.max(0)[pcs], X_train.shape[1],
                                      feature_ids=pcs)
        # Paint the decision boundary
        cmap = colors.ListedColormap(['lightcoral', 'mediumseagreen'])
        ax.pcolormesh(xx, yy, zz.astype(float), cmap=cmap)
        clrs = [ ('lightcoral', 'mediumseagreen')[int(y_i)] for y_i in y_train]
        ax.scatter(X_train.values[:, pcs[0]], X_train.values[:, pcs[1]],
                   ec='k', color=clrs, s=25)
        ax.set_title(f'C: {c_val}, kernel: {ker}')
        ax.set_xlabel(f'PC{pcs[0]}')
        ax.set_ylabel(f'PC{pcs[1]}')
        plot_idx += 1
plt.show()
```

Listing 25: Code I used to create a 6x4 plot grid for each decision boundary of the training stage

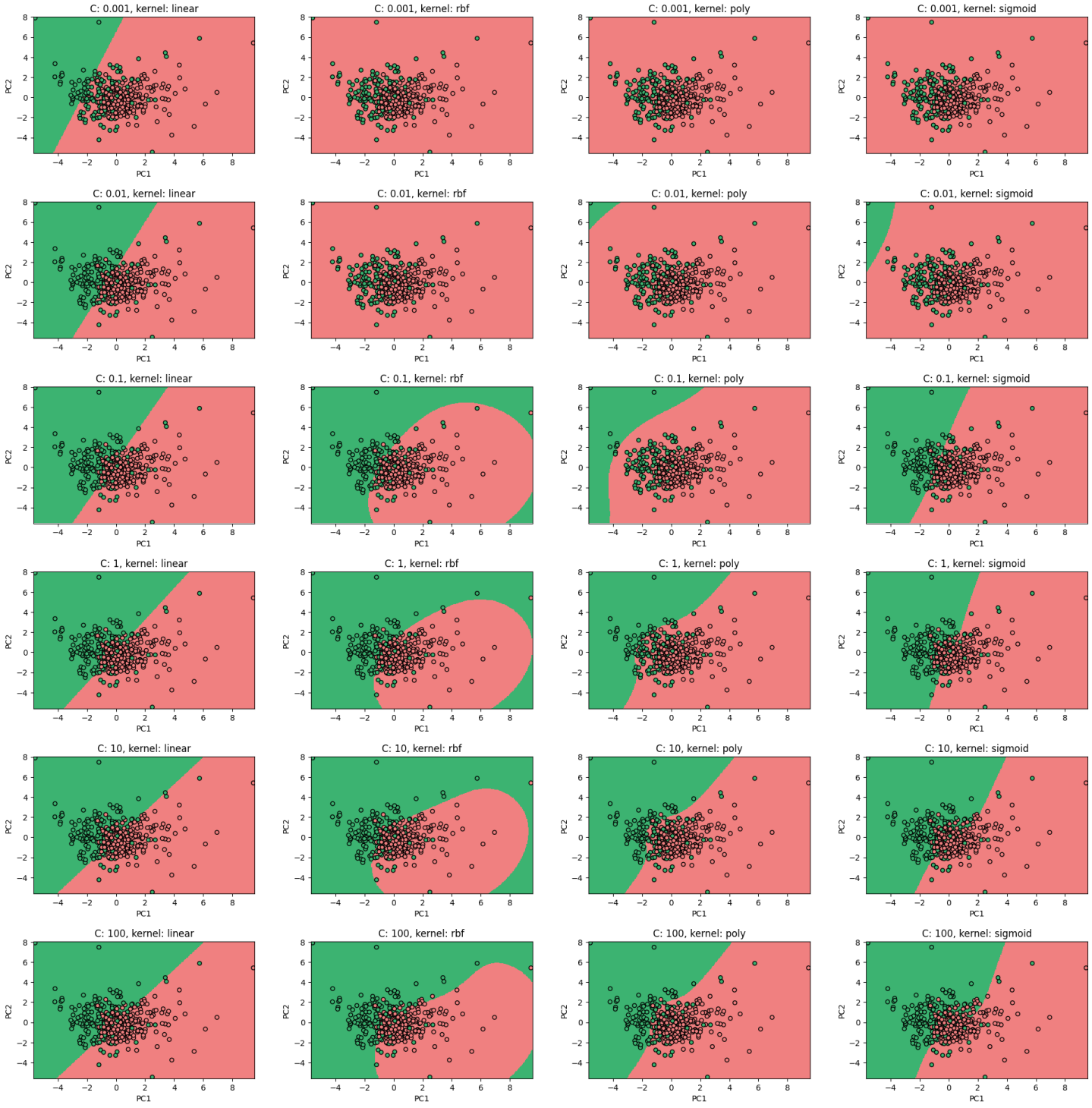

Figure 24: Every decision boundary for each combination of C value and Kernel explored in the training stage

1. C = 0.001: For all kernels, the decision boundary does not classify the points effectively, suggesting that the regularization is too strong (over-regularized), leading to high bias and potentially high error on the training set.

2. C = 0.01: Improvement can be seen, especially with the RBF kernel which starts to show some non-linearity. However, linear, poly, and sigmoid still seem to underfit.

3. C = 0.1:

   - The RBF kernel shows clear non-linearity and does a decent job in classifying points.
   - Linear kernel, however, remains mostly unchanged and still underfits.
   - Poly and Sigmoid kernels show more flexibility than before, but might still be underfitting.

4. C = 1:

   - RBF kernel starts to fit the data well, capturing the inherent distribution.
   - Linear kernel remains similar, underfitting the data.
   - Poly kernel shows improvements but still might be slightly underfitting.
   - Sigmoid kernel appears to struggle to capture the data's distribution.

5. C = 10:

   - RBF kernel fits the data almost perfectly, with the potential danger of overfitting.
   - Linear kernel remains largely the same.
   - Poly kernel improves further, though still underfits slightly.
   - Sigmoid kernel still seems inadequate for this data.

6. C = 100:

   - RBF kernel continues its trend, and there's a higher risk of overfitting.
   - Linear, Poly, and Sigmoid do not provide significant improvements.

In summary, linear Kernel remains consistent across different C values and tends to underfit the data. The rbf Kernel performs best for this data, especially at higher C values, but risks overfitting with very high C.The poly Kernel improves with increasing C, but never quite captures the data distribution perfectly.An finally, the sigmoid Kernel appears to be the least effective for this data across all C values.

7.3 Testing

The SVM model will be implemented using a linear kernel and $C = 1$ value26.

```
# Initialize the Decision tree classifier and fit
X_train, y_train = bcw_train.drop('target', axis=1), bcw_train['target']

svc_best = SVC(C=1, kernel='linear')
svc_best.fit(X_train.iloc[train], y_train.iloc[train])

X_test, y_test = bcw_test.drop('target', axis=1), bcw_test['target']
y_pred = svc_best.predict(X_test)

final_metrics = pd.DataFrame(data={'Accuracy': accuracy_score(y_test, y_pred),
                                   'Precision': accuracy_score(y_test, y_pred),
                                   'Recall': recall_score(y_test, y_pred),
                                   'F1-Score': f1_score(y_test, y_pred)}, index=[0])
final_metrics['Model'] = 'SVM'
# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)
# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Listing 26: Code snippet for SVM testing

7.3.1 Final performance evaluation

	Accuracy	Precision	Recall	F1-Score
0	0.979	0.979	0.985	0.971

Table 16: Final performance metrics

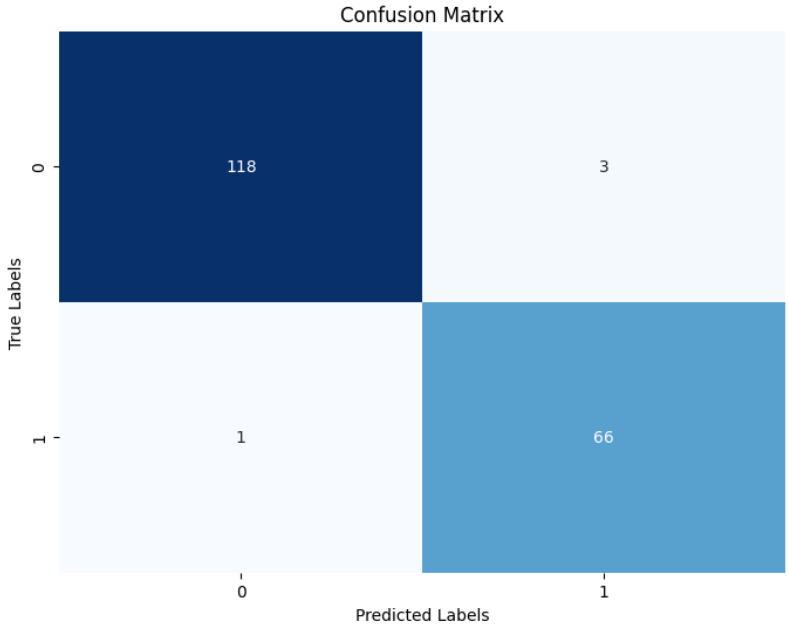

Figure 25: Confusion matrix for SVM

The final test metrics reveal an outstanding performance of the model on the test data.

With an accuracy of 97.9%, the model correctly predicted the majority of the instances. Precision, which measures the correctness of the positive predictions, is at 97.9%. This means that out of all the instances the model predicted as positive, 97.9% were indeed positive. Recall stands at 98.5%. This metric indicates that the model identified 98.5% of all actual positive instances. Lastly, the F1-Score is at 97.1%. This score suggests a well-balanced model, as it equally considers both false positives and false negatives.

Overall, the model demonstrates robust and reliable performance across all metrics.

I finally include the decision boundary for the tuned model:

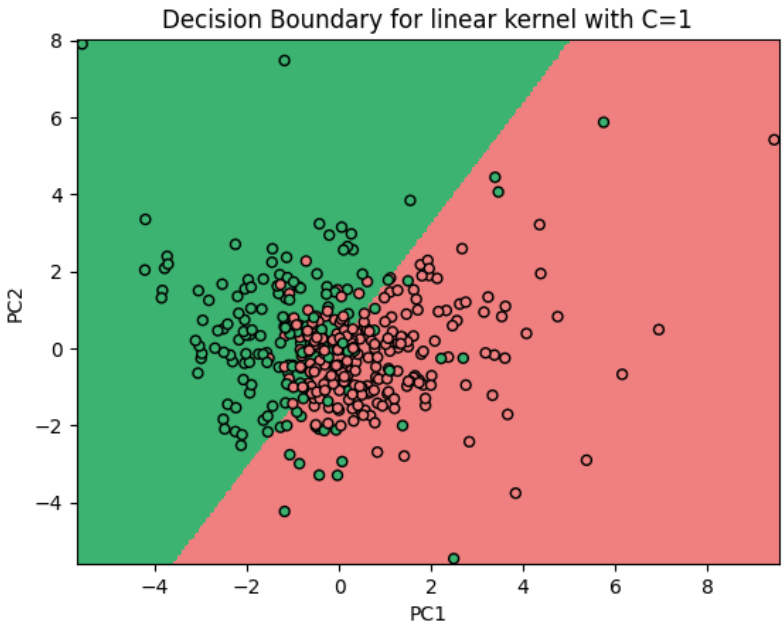

Figure 26: Decision boundary for the fine tuned model

8 Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is a type of feedforward artificial neural network consisting of at least three layers: an input layer, one or more hidden layers, and an output layer.

Each layer is made up of neurons, with the neurons in adjacent layers connected by weighted edges. The network processes input data in a single direction, from the input layer through the hidden layers to the output layer, without any recurrent or feedback loops.

In order to effectively describe how a MLP works and fine tune the model to make predictions on the input data im going to break down the model into the concepts that its made from and describe them one by one.

8.1 Structure of a Multi-Layer Perceptron model

Neuron

The neuron is the atomic unit of Artificial Neural Networks (ANN). It takes a set of inputs x_i, applies a linear transformation which involves multiplying the input by a weight w_i and adding a bias b. Finally it passes the result through an activation function which introduces a non-linearity to the Neural Network (NN).

$$O = \text{Activation}\left(\sum_{i=1}^{n}(x_i \cdot w_i) + b\right)$$

Figure 27: Mathematical output of a neuron "O", where n is the number of inputs

The *weights* are the coefficients applied to the inputs to control the influence of each input on the neuron's output. They're adjusted during the training process to minimize the loss function in order to improve the model predictions on a given set of data.

The *bias* term allows the neuron to have some flexibility, enabling it to activate even when the weighted sum of inputs is not sufficient. Essentially, it shifts the activation function to the left or right, helping the model fit the data better.

The *activation function* in a neural network plays a critical role in introducing non-linearity into the system. This allows the model to capture complex relationships between inputs and outputs, which is essential for learning from real-world data that's rarely linearly separable or representable.

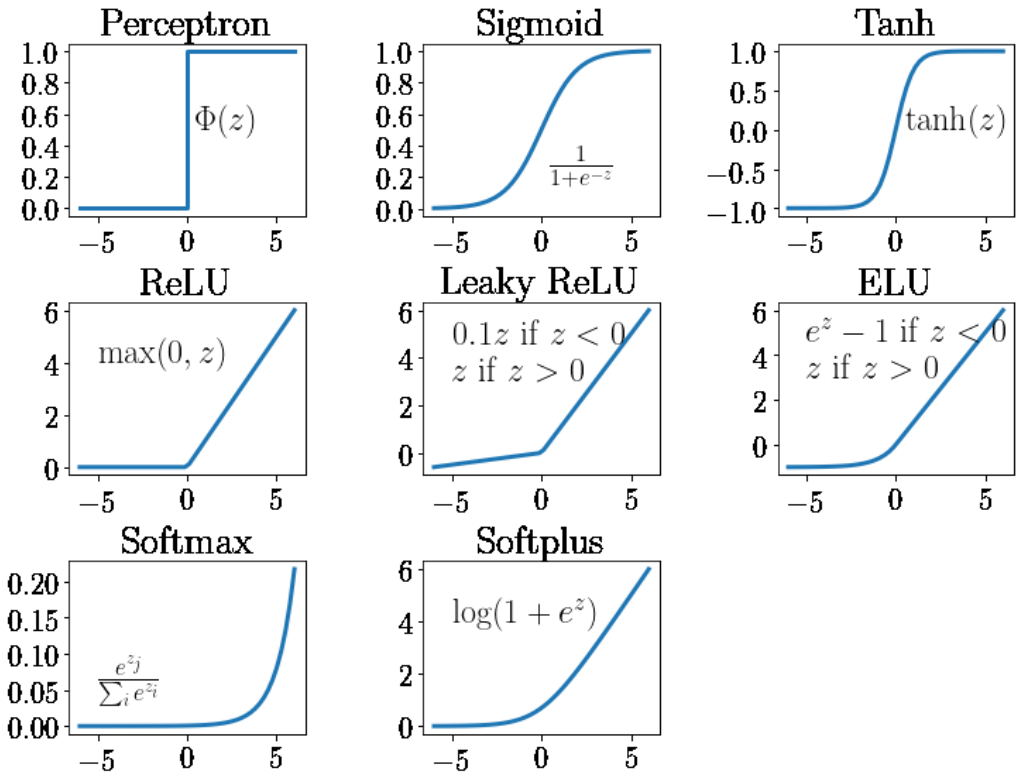

Figure 28: Different activation functions [7]

Function	**Equation**	**Range**	**Properties**
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	[0, 1]	Vanishing gradient
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	[-1, 1]	Vanishing gradient, zero-centered
ReLU	$f(x) = \max(0, x)$	[0, ∞]	Mitigates vanishing gradient, dead neurons
Leaky ReLU	$f(x) = \max(\alpha x, x)$	($-\infty$, ∞)	Non-zero gradient for $x < 0$
Softmax	$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	(0, 1)	Converts scores to probabilities

Table 17: Caption

Input Layer

Each neuron in the input layer typically corresponds to one feature in the dataset. Each of these neurons would receive a single feature value for a given sample as its input during the forward pass of the network.

This one-to-one mapping between features and input neurons allows the network to take in the entire feature vector for each sample in the dataset. The values are then propagated through the hidden layers and finally to the output layer, where a prediction is made.

Hidden Layers

Neurons organize in layers, each neuron of the previous layer is connected to every neuron on the next layers (this created a *fully-connected layer*). The amount of layers that a MLP accepts goes from 0 (then it'd be called a *Perceptron*) and 3. More than 3 layers would make the model fall into the category of *Deep Learning (DL)* rather than ML.

Output Layer

Produces the final prediction or classification. The number of neurons here corresponds to the number of classes in a classification problem or just one neuron for regression problems.

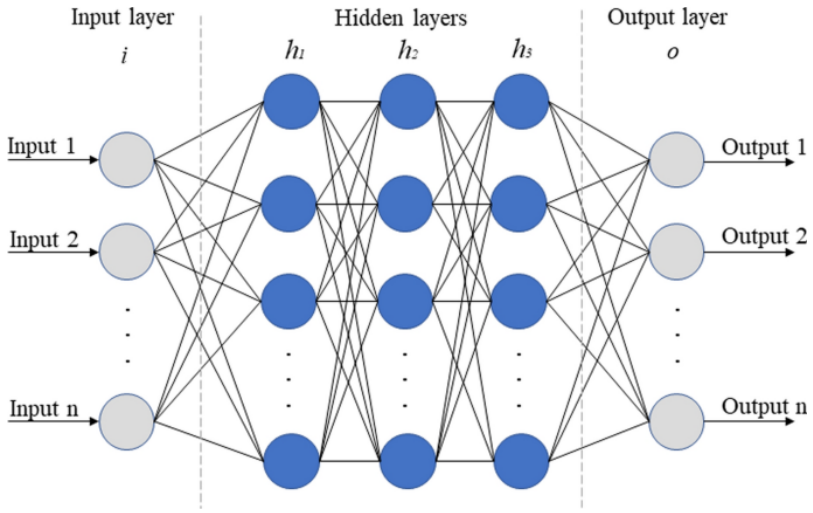

Figure 29: Typical MLP structure. The input layer has one neuron per feature in the dataset. Three hidden layers. The output layer consists of one neuron per predicted class. [8]

8.2 Learning process of Artificial Neural Networks

In the context of an artificial neural network (ANN), the output layer produces an estimated value $\hat{y}$ based on the given input. The fidelity of this estimate to the actual target value may vary. To enhance the model's predictive accuracy, it is essential to employ an algorithm for optimizing the model's parameters, the weights w_i and biases b_i.

Backpropagation is an optimization algorithm used for reducing the error in the predictions of a neural network (like an MLP). It is a type of supervised learning algorithm that uses the gradient descent or variations of it (like stochastic gradient descent) to minimize the loss function.

In order to understand how backpropagation fits the parameters of our NN to the input data we must break into 4 steps the propagation process of our ANN:

Forward Pass

Input data is passed forward through the network. Each hidden layer accepts the input data, processes it as per the activation function, and passes it to the next layer.

Compute Loss

The prediction at the output layer is compared to the actual target value, using a *loss function* to measure the error. The loss function is different for each backpropagation algorithm. Two popular loss functions are the Mean Squared Error (MSE) (figure 30) for regression tasks and the Cross-Entropy Loss (figure 31) for classification tasks.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Figure 30: Mean Squared Error

$$\text{Cross-Entropy} = -\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Figure 31: Cross-Entropy Loss

Backward Pass

The gradient of the loss function with respect to each weight and bias is computed. This involves computing the derivative of the loss function and propagating it backward through the network. The core mathematical tool for this is the chain rule of calculus.

Let's say we have a neural network with one hidden layer, and we are currently at that hidden layer. Let f be the activation function, z the input to the neuron before activation, w the weight, x the input from the previous layer, b the bias, and $\mathcal{L}$ the loss function. The output of the neuron is $f(z)$ (see Equation 32)

$$z = w \cdot x + b$$

$$f(z) = f(w \cdot x + b)$$

$$\mathcal{L} = \text{Loss}(f(z), y)$$

Figure 32: Relationship between ANN variables, y is the actual target value.

For the **weights** we want to find $\frac{\partial \mathcal{L}}{\partial w}$, the gradient of the loss function with respect to the weight w.

By the chain rule, we have:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial f} \times \frac{\partial f}{\partial z} \times \frac{\partial z}{\partial w}$$

1. $\frac{\partial \mathcal{L}}{\partial f}$: This is the gradient of the loss with respect to the output of the neuron. It essentially tells us how the loss changes with respect to the final output.
2. $\frac{\partial f}{\partial z}$: This is the derivative of the activation function. It tells us how the output of the neuron changes with a change in its input z.
3. $\frac{\partial z}{\partial w}$: This tells us how the weighted sum z changes with a change in the weight w. This is equal to the input x from the previous layer.

For the **biases** we proceed using the same logic.

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial f} \times \frac{\partial f}{\partial z} \times \frac{\partial z}{\partial b}$$

1. $\frac{\partial \mathcal{L}}{\partial f}$ is the gradient of the loss with respect to the output of the neuron.
2. $\frac{\partial f}{\partial z}$ is the derivative of the activation function.
3. $\frac{\partial z}{\partial b}$ is the derivative of the input to the neuron with respect to the bias. This is simply 1 since $z = w \cdot x + b$ and $\frac{\partial (w \cdot x + b)}{\partial b} = 1$.

This process is repeated for each layer, moving from the output layer back to the input layer, thereby "propagating" the error backwards through the network.

The **key concept** in this section is that the gradient serves as an indicator of how the loss function varies. A low gradient value suggests proximity to a local minimum of the loss function, indicating that the neural network is nearing the conclusion of its training phase. In this stage, the model parameters(weights and biases) are nearly optimized. Given that the gradient is low, the subsequent updates to these parameters will also be minimal.

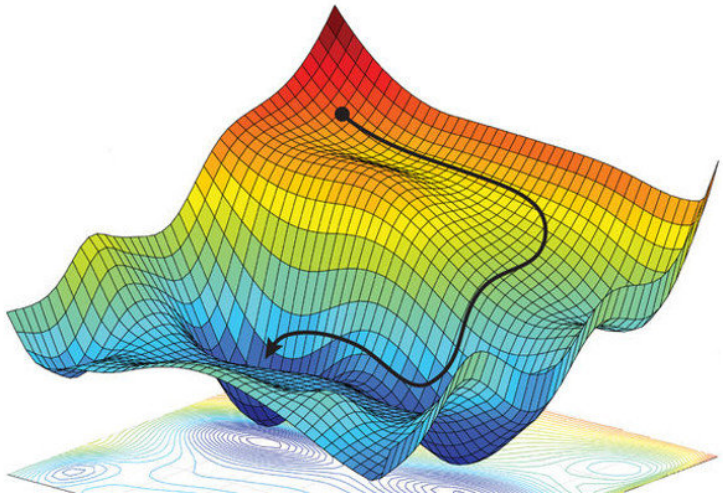

Figure 33: Loss function for an oversimplified MLP, the x and $y-axis$ correspond to two weights w_1, w_2 and the $z-axis$ corresponds to the loss function value for those weights. The black line in the figure indicates the starting point (with a black dot) and the end point (with an arrow). The end point corresponds to a local minimum of the function, suggesting that the tuning for the ANN parameters would have been successful. [9]

Update Weights and Biases

Once the gradient is computed, the weights and biases are updated in the opposite direction of the gradient (since we want to find a local minimum). This is usually done using gradient descent or its variants.

$$w_{\text{new}} = w_{\text{old}} - \alpha \times \frac{\partial \mathcal{L}}{\partial w}$$

$$b_{\text{new}} = b_{\text{old}} - \alpha \times \frac{\partial \mathcal{L}}{\partial b}$$

Figure 34: New weight and bias values, computed using the gradient descent method. Here, α is the learning rate, and it's a hyperparameter we must fine tune in the code.

During this update stage, certain techniques can be incorporated to optimize the model fit on the input data while preventing overfitting and underfitting. Two such techniques are known as L1 and L2 regularization. When you include L1 and/or L2 regularization, the gradient of the regularization term must also be considered when updating each weight.

L1 Regularization

L1 regularization introducen a the term shown in ecuation 35 to each weight and bias when updating them. Since this term in introduced in the gradient decent stage, we must compute the derivative of the L1 term with respect to w.

$$\text{L1} = \lambda_1 \sum_{i=1}^{n} |w_i|$$

$$w_{\text{new}} = w_{\text{old}} - \alpha \left(\frac{\partial L}{\partial w} + \lambda_1 \text{sign}(w) \right)$$

Figure 35: L1 regularization term and the new weights with L1 term.

L2 Regularization

The procedure is the same, but the L2 regularization term is different (shown in ecuation 36)

$$\text{L2} = \lambda_2 \sum_{i=1}^{n} w_i^2$$

$$w_{\text{new}} = w_{\text{old}} - \alpha \left(\frac{\partial L}{\partial w} + 2\lambda_2 w \right)$$

Figure 36: L2 regulatization term and the new weights with L2 term.

L1 and L2 regularization modify the weight update rule by adding additional terms to the gradients. This helps in penalizing large weights, thereby making the model more robust and preventing overfitting.

8.3 Model Training

8.3.1 Training and validation

Since there are a lot of hyperparameters to fine-tune, I will start with a random/"standard" MLP configuration:

```
MLPClassifier(hidden_layer_sizes=(20,), batch_size=50,
              learning_rate_init=1e-3, earning_rate='constant',
              shuffle=True, solver='adam', random_state=42)
```

Listing 27: MLP that i randomly chose

- Hidden Layer Sizes (`hidden_layer_sizes`). The model is set to have a single hidden layer with 5 neurons. The tuple `(5,)` specifies the number of neurons in each hidden layer.
- Batch Size (`batch_size`). The model will use mini-batch gradient descent with a batch size of 50. This means that for each iteration of training, 50 samples will be used to compute the gradient and update the model's weights.
- Learning Rate Initialization (`learning_rate_init`). The initial learning rate for the weight updates is set to `1e-3` (or 0.001). The learning rate controls how big the weight updates are during training. A smaller learning rate means slower convergence but potentially better performance, while a larger learning rate means faster convergence but potentially overshooting the optimal solution.
- Learning Rate Policy (`learning_rate`). It's set to `'constant'`, which means the learning rate will remain unchanged throughout the training process.
- Shuffling (`shuffle`). With `shuffle=True`, the training data will be shuffled at each iteration to prevent cycles.
- Solver (`solver`). The optimizer used is `'adam'`. Adam is a popular optimization algorithm that combines the benefits of two other extensions of stochastic gradient descent: AdaGrad and RMSProp.
- Random State (`random_state`). The random seed is set to 10. This is used for reproducibility purposes.

```
fig, axes = plt.subplots(nrows=10, ncols=2, figsize=(10,30))
fig.tight_layout(pad=4.0)

for i, (train, validation) in enumerate(scv.split(X, y)):
    # Initialize and train MLP
    mlp = MLPClassifier(hidden_layer_sizes=(5, ), batch_size=50,
                        learning_rate_init=1e-3, learning_rate='constant',
                        shuffle=True, solver='adam', random_state=42)
    mlp.fit(X.iloc[train], y.iloc[train])
    y_pred = mlp.predict(X.iloc[validation])

    # Plot Decision Boundary on the left column
    xx, yy, zz = decision_boudary(mlp, X.min(axis=0)[pcs],
                                  X.max(axis=0)[pcs], X.shape[1],
                                  feature_ids=pcs)
    axes[i, 0].contourf(xx, yy, zz, alpha=0.4)
    axes[i, 0].scatter(X.iloc[:, 0], X.iloc[:, 1], c=y,
                       edgecolors='k', marker='o')
    axes[i, 0].set_title(f'Fold {i} Decision Boundary')

    # Plot Loss curve on the right column
    axes[i, 1].plot(mlp.loss_curve_)
    axes[i, 1].set_title(f'Fold {i} Loss Function')
    axes[i, 1].set_xlabel('Iterations')
    axes[i, 1].set_ylabel('Loss')

    # Store performance metrics
    metrics_df.loc[f'Fold {i}', 'Accuracy'] = accuracy_score(
                                              y.iloc[validation], y_pred)
    metrics_df.loc[f'Fold {i}', 'Precision'] = precision_score(
                                               y.iloc[validation], y_pred)
    metrics_df.loc[f'Fold {i}', 'Recall'] = recall_score(
                                            y.iloc[validation], y_pred)
    metrics_df.loc[f'Fold {i}', 'F1-Score'] = f1_score(
                                              y.iloc[validation], y_pred)
```

Listing 28: Trainig with SCV for the MLP model

8.3.2 Performance in training stage and fine-tuning hyperparameters

By executing the previous code we get the Image

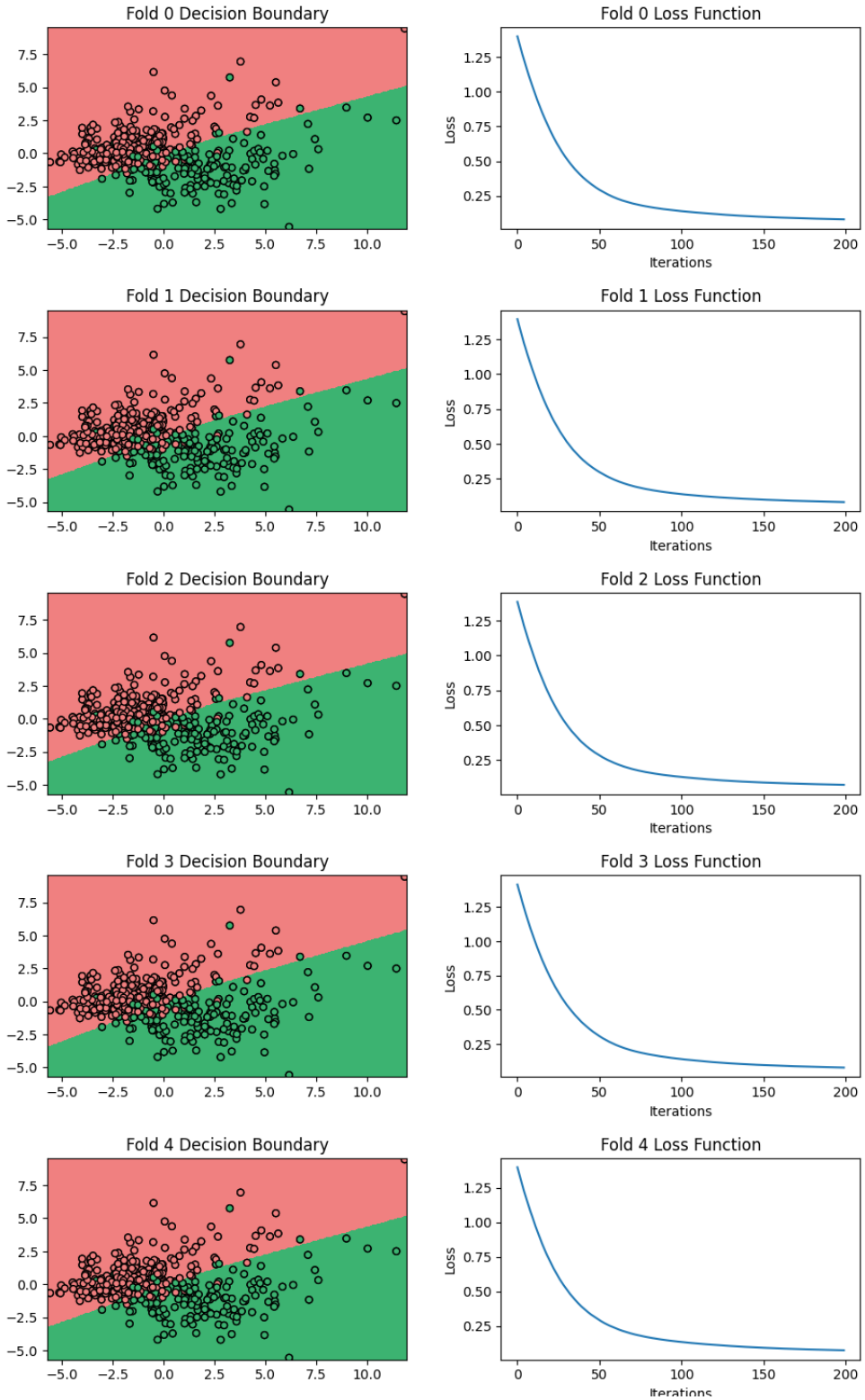

Figure 37: Decision boundary and Loss function variation for each fold (I)

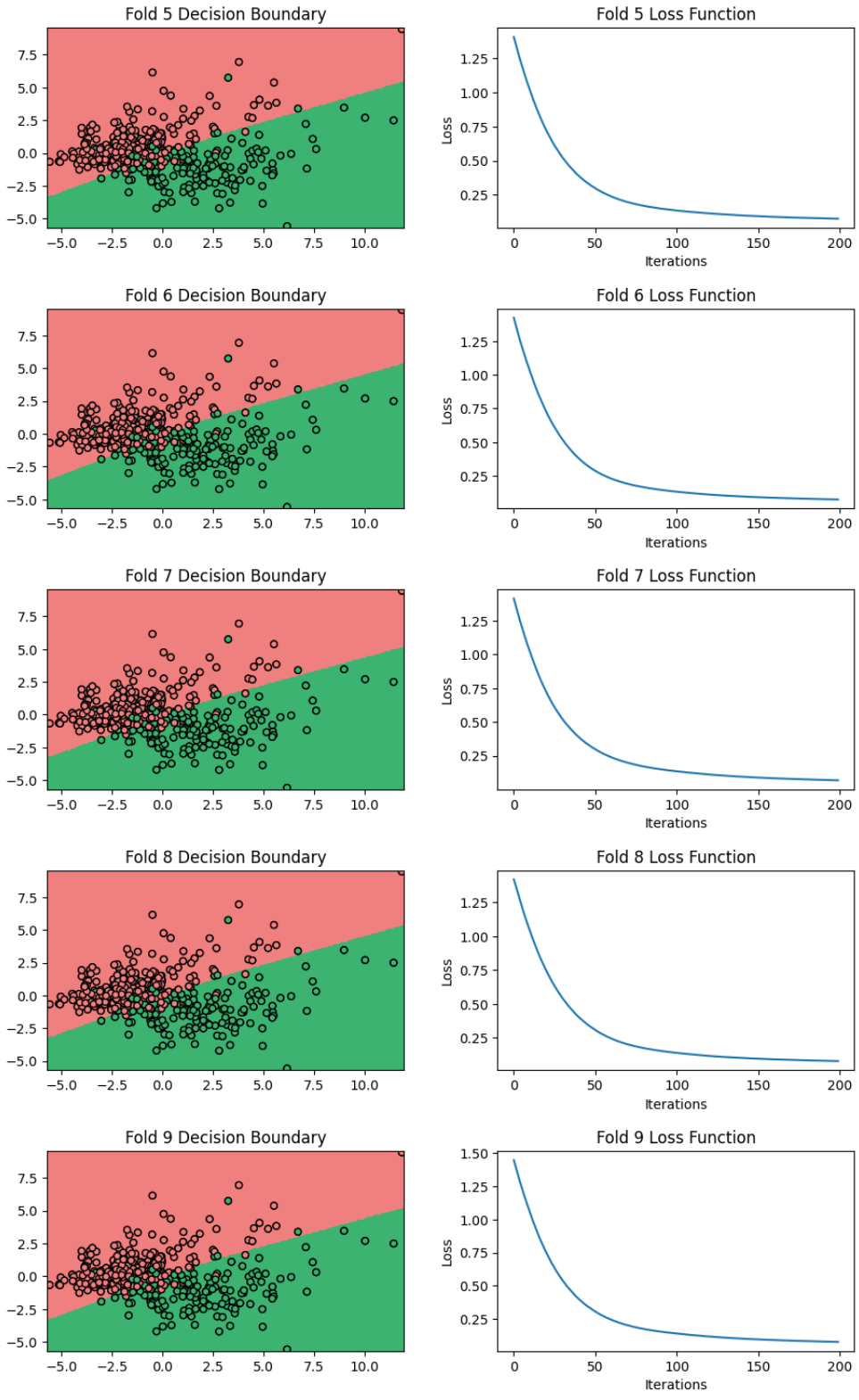

Figure 38: Decision boundary and Loss function variation for each fold (II)

	Accuracy	Precision	Recall	F1-Score
Fold 0	0.974	1.000	0.933	0.966
Fold 1	0.974	1.000	0.933	0.966
Fold 2	0.974	1.000	0.933	0.966
Fold 3	0.974	1.000	0.933	0.966
Fold 4	0.974	1.000	0.933	0.966
Fold 5	0.974	1.000	0.929	0.963
Fold 6	0.921	0.923	0.857	0.889
Fold 7	0.974	1.000	0.929	0.963
Fold 8	0.974	1.000	0.929	0.963
Fold 9	0.947	0.929	0.929	0.929

Table 18: SML metrics for each fold for the MLP model

Upon observation of the table 18 we can see that our model scored almost perfectly for each metric, furthermore, since the MLP is a flexible model with a many hyperparameters to tune, there is some room for improvement.

I will proceed by tuning the number of hidden layers and the number of neurons per layer.

Fine Tuning the number of hidden layers and neurons per layer

In order to effectively fine tune these two hyperparameters I will provide a visualization of the decision boundary and loss function change accompanied by the accuracy and f1-score ML performance measures.

The code is really similar to the previous section. For each fold I train the model with a number of hidden layers and neurons per layer.

I merge every loss function into one array to later divide each value of that array by the number of folds 28 in order to compute the average loss function.

```
for layer_configs in hidden_layer_configs:
    for hidden_layer_sizes in layer_configs:
        # Initialize data containers for combined plots
        combined_loss_curve = []
        (...)
        for i, (train, validation) in enumerate(scv.split(X, y)):
            (...)
            # Combine loss curves
            if i == 0:
                combined_loss_curve = np.array(mlp.loss_curve_)
            else:
                # Ensure the length of loss_curve_ is
                # the same for accumulation
                min_len = min(len(combined_loss_curve),
                              len(mlp.loss_curve_))
                combined_loss_curve[:min_len] += np.array(
                                                 mlp.loss_curve_[:min_len])

        # Average the combined_loss_curve
        combined_loss_curve /= 10 # 10 folds
        (...)
```

Listing 29: Trainig with SCV for the MLP model

For each fold I save the accuracy and f1-score in order to compute the mean of that values and then store them into a pandas dataframe to consult later.

Finally, in order to find the best trade off between computational cost, model fidelity and complexity of the ANN, I will use the library `time` to measure how much time took to train using SCV with 10 folds each of the NN configurations we are going to try.

These are the results:

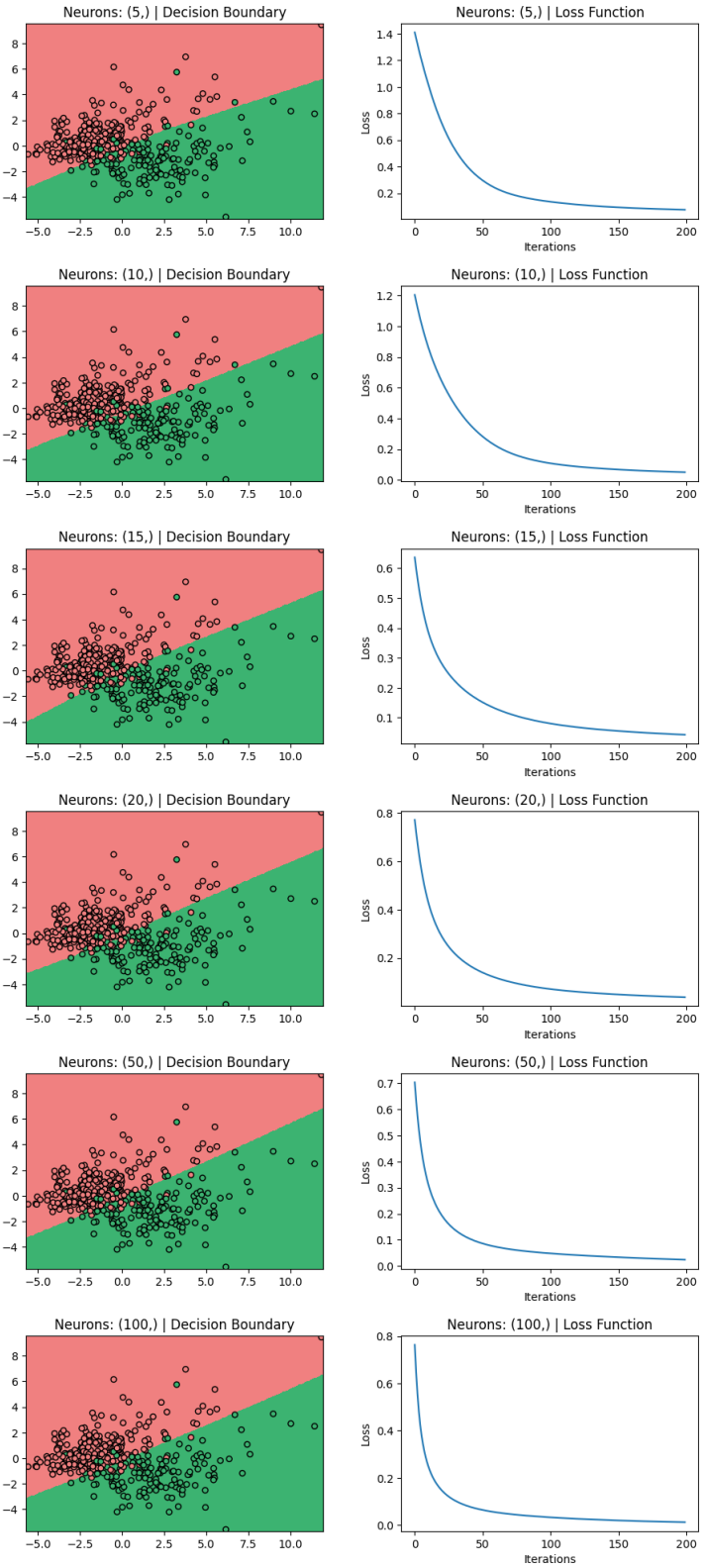

Figure 39: One hidden layer results

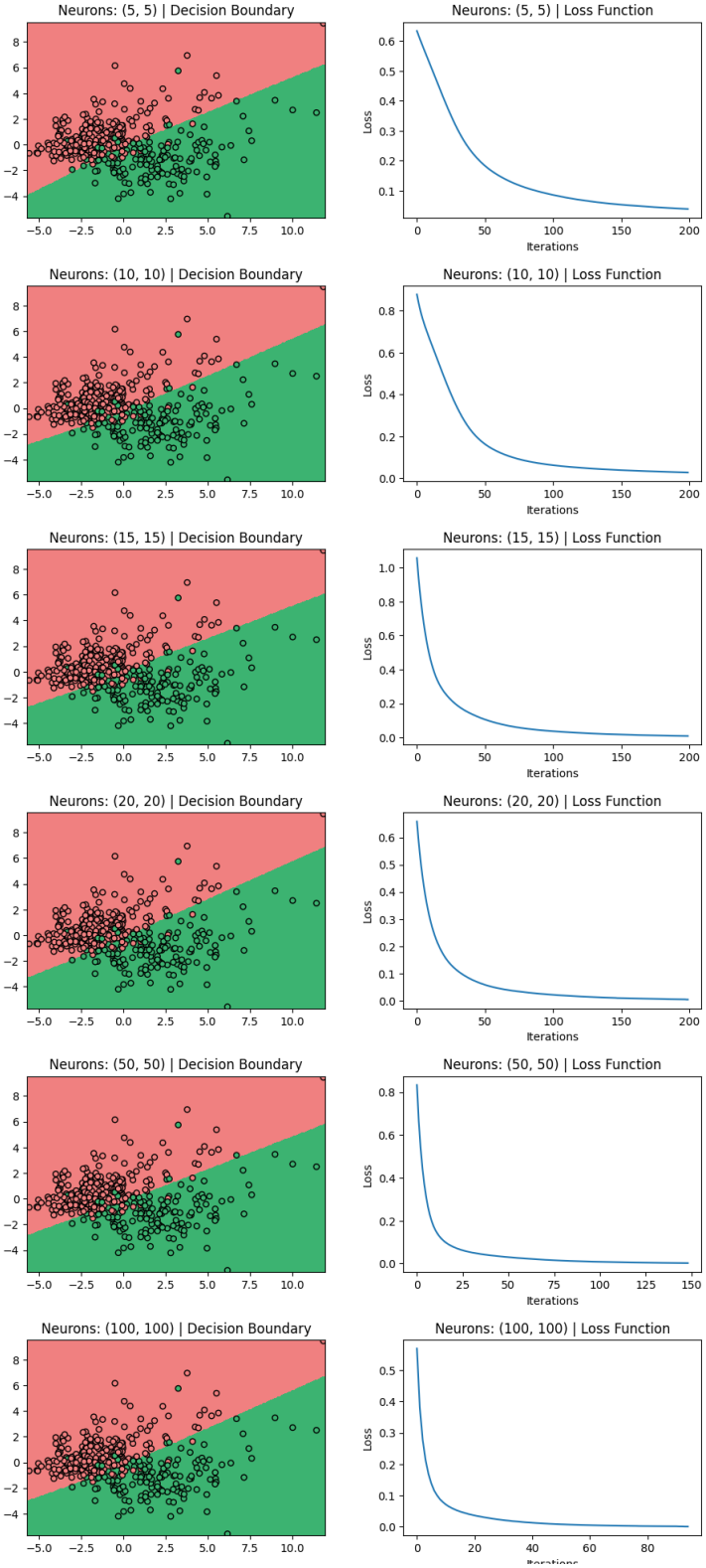

Figure 40: Two hidden layers results

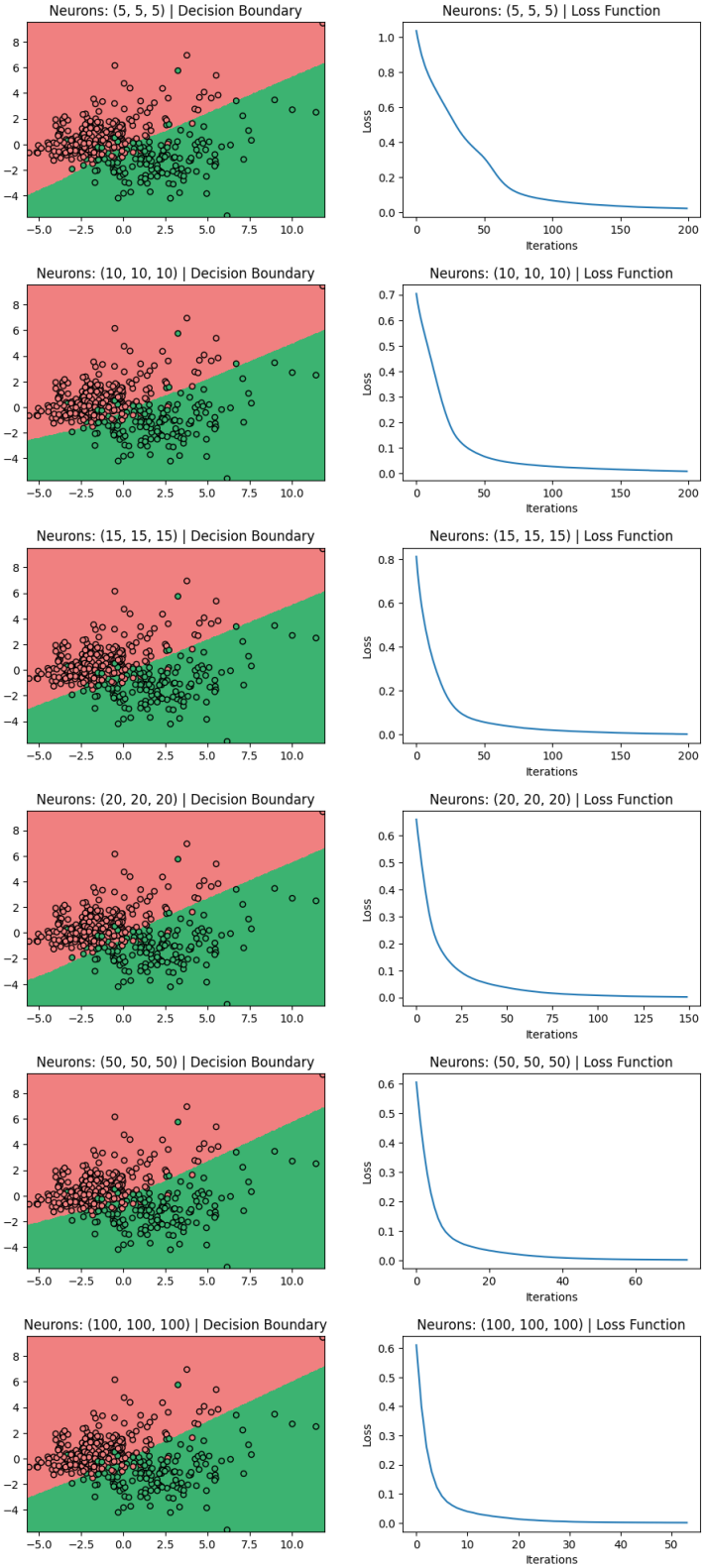

Figure 41: Three hidden layers results

	Hidden layers	Neurons per Layer	Accuracy	F1-Score	CPU Time
0	1.000	5.000	0.966	0.953	4.362
1	1.000	10.000	0.974	0.964	5.130
2	1.000	15.000	0.968	0.958	5.561
3	1.000	20.000	0.982	0.975	4.469
4	1.000	50.000	0.982	0.975	6.418
5	1.000	100.000	0.976	0.968	6.348
6	2.000	5.000	0.968	0.958	6.476
7	2.000	10.000	0.969	0.957	6.866
8	2.000	15.000	0.979	0.972	6.008
9	2.000	20.000	0.984	0.978	7.656
10	2.000	50.000	0.974	0.964	5.897
11	2.000	100.000	0.982	0.975	11.063
12	3.000	5.000	0.976	0.968	8.851
13	3.000	10.000	0.969	0.957	8.161
14	3.000	15.000	0.966	0.955	8.780
15	3.000	20.000	0.969	0.957	5.804
16	3.000	50.000	0.969	0.957	5.146
17	3.000	100.000	0.984	0.978	7.859

Table 19: Data related to the MLP performance for each hidden layer and neuron per layer

In order to get the useful data from table 19, I've run the code listed in 30, the output of this code is shown in 20.

```
results_df.iloc[results_df['Accuracy'].idxmax(), :]
results_df.iloc[results_df['F1-Score'].idxmax(), :]
```

Listing 30: Pandas code to filter the dataset

Indexes	17	9
Hidden layers	3.000	2.000
Neurons per Layer	100.000	20.000
Accuracy	0.984	0.984
F1-Score	0.978	0.978
CPU Time	7.859	7.656

Table 20: Highest Accuracy and highest F1-Score

Upon observation of table 20, we can conclude that even though the value of accuracy and f1-score is the same for the two ANN architectures, the NN with 2 hidden layers and 20 neurons per layer executes in less time and is less complex, therefore, we will proceed by evaluating some other hyperparameters for this architecture.

Also, if we take a look again at table 20, we can stubble upon some strange data, since for 100 neurons per layer and 2 layers the amount of CPU time invested in training this model is higher than other complex architectures (such as 3 layers with 100 neurons per layes). This might be because of the Google Collab IDE and other internal resources Python has to fetch, this resources would add an offset to the CPU time.

Before proceeding with the rest of the fine-tuning, I wanted to provide some visuals on how the SML performance measures behave for different number of neurons (see Figures 42).

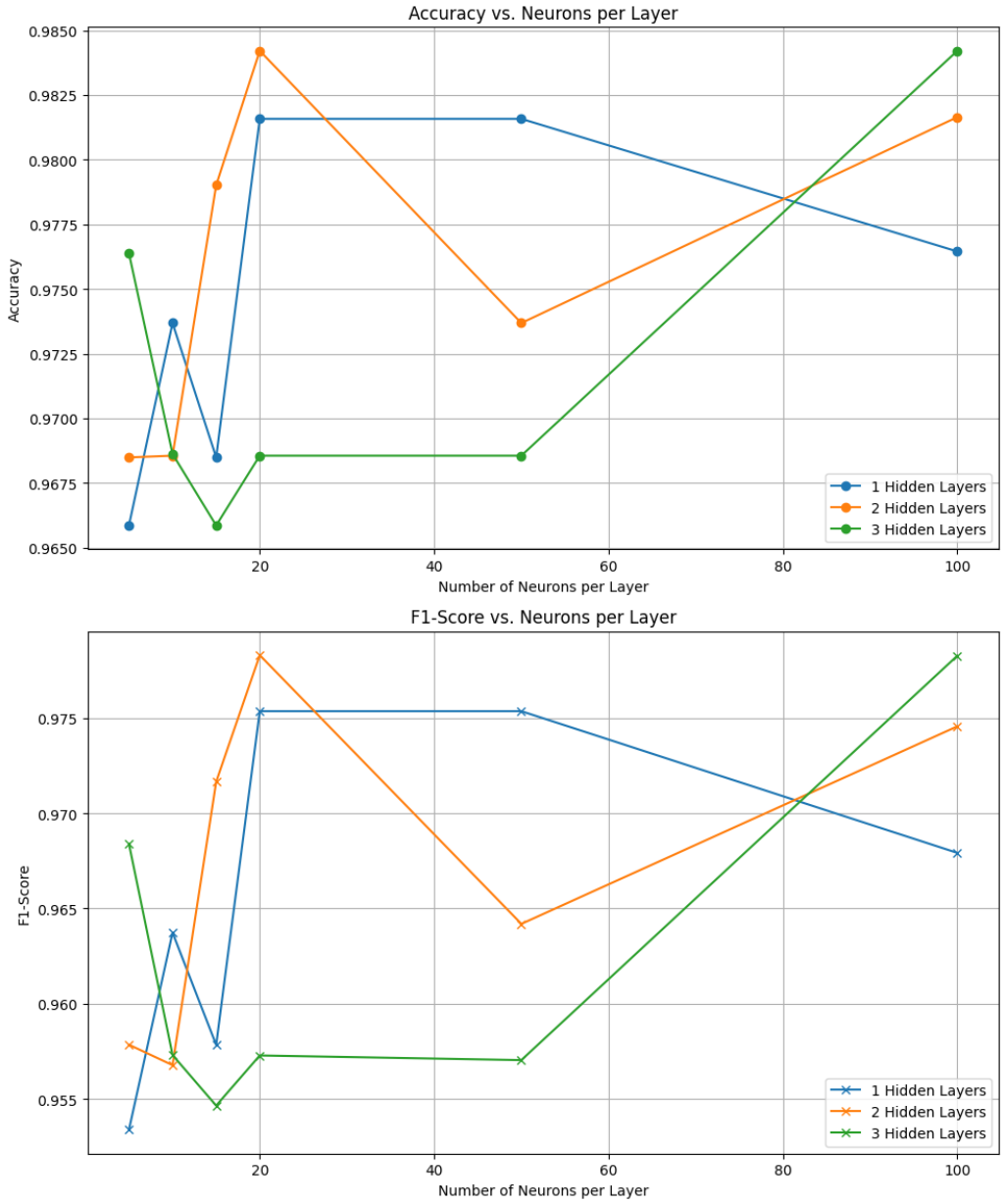

Figure 42: Accuracy and F1-Score variation when incrementing the number of neurons per layer for different amounts of hidden layers

Fine Tuning the Learning Rate

The code for this section is too large and I'll include it in the GitHub repository

linked at the end of the document. Basically, we train a model with `(20, 20)` NN architecture and a number of learning rates. For each learning rate value I'll display a *decision boundary* (Figure 43), the *cross-entropy value 31 for the output* and the *accuracy and f1-score* (Figure 44).

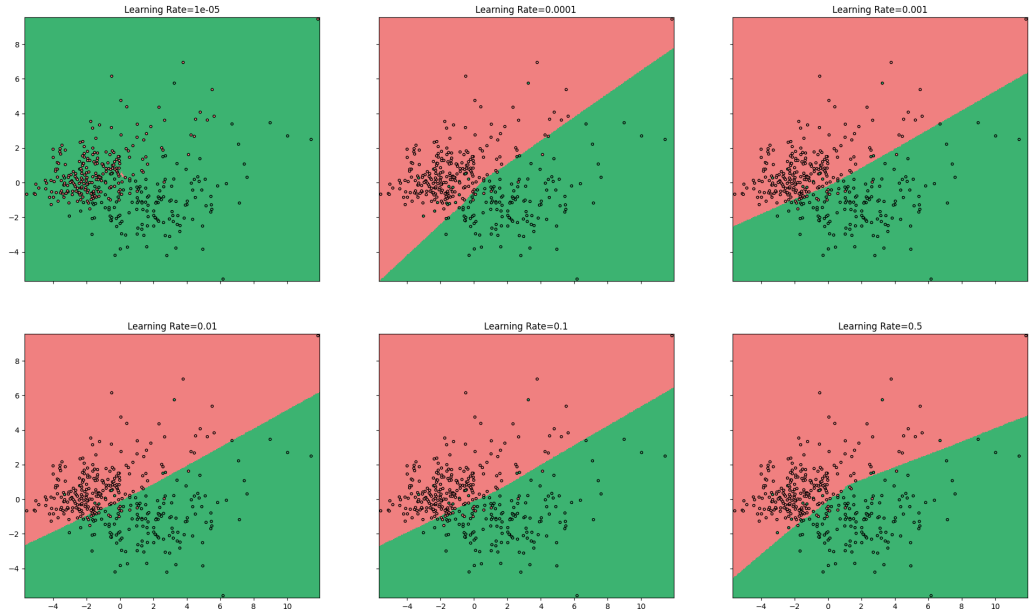

Figure 43: Decision boundary for each learning rate. As always, the X axis is the PC1 and Y axis is PC2, but the axis label did not show.

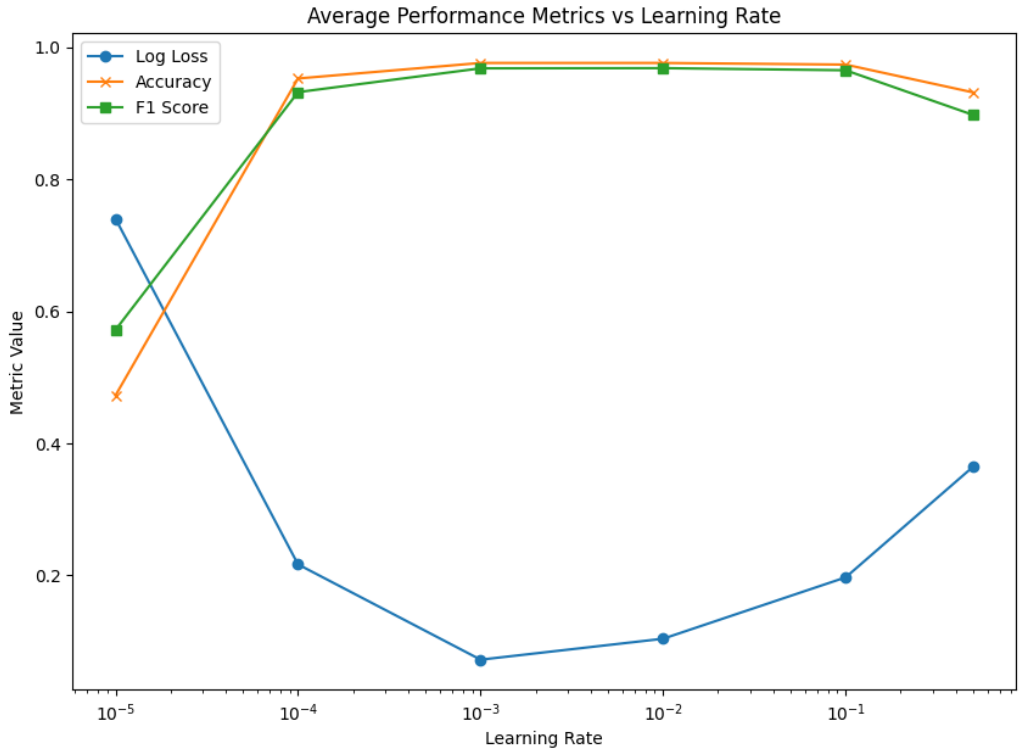

Figure 44: Cross Entropy Log Loss, Accuracy and F1-Score for the MLP.

Upon observation of the Figures 44 and 43, we can conclude that the learning rate value that minimizes cross entropy log loss while maximizing accuracy and f1-score is 10^{-3} (`1e-3`). Which was the value previous to the exploration.

Fine Tuning the Alpha value for L2 Regularization

As we commented above, the alpha value in an MLP determines the regularization strength applied to the network, which helps in preventing overfitting.

Regularization introduces a penalty on the magnitude of the parameters, encouraging them to stay small. By exploring various alpha values, we aim to find an optimal balance between the model's ability to fit the training data and its generalization to unseen data.

This is critical, especially in complex models like neural networks, where without proper regularization, the model might simply memorize the training data without capturing the underlying patterns, leading to poor generalization on test or real-world data.

To fine tune this parameter I will follow the same workflow as before, the code will be really similar and It will be attached in the GitHub repository at the end of the document.

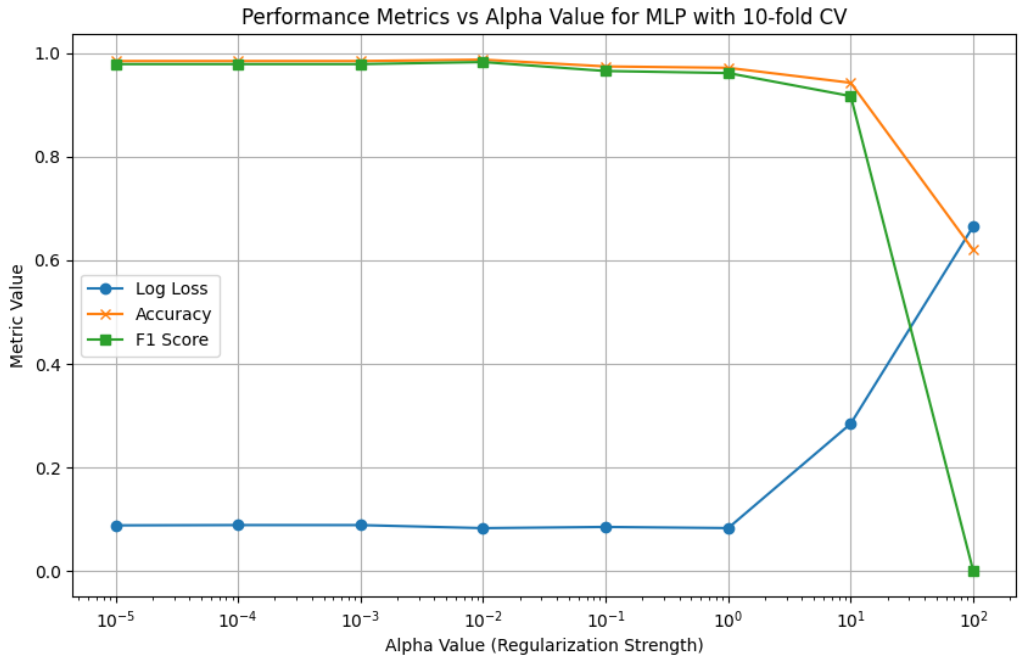

Figure 45: Cross Entropy Log Loss, Accuracy and F1-Score for each alpha value.

From the Figure 45, we can observe that as the alpha value increases, the Log Loss decreases until a certain point, after which it spikes dramatically.

This indicates that for really small alpha values, the model might be overfitting, but as the regularization strength increases, (from `1e-3` to `1e-1`) it begins to perform optimally, minimizing the loss. However, after a certain threshold, the regularization becomes too strong, causing the model to underfit and therefore increasing the Log Loss.

Both Accuracy and F1 Score maintain a high and stable performance for lower alpha values but experience a sharp decline when the alpha becomes too large. An ideal alpha value would be one that minimizes the Log Loss without causing a substantial drop in Accuracy or F1 Score, suggesting that in this case, an alpha value in the range of `1e-3` to `1e-2`) might be optimal.

8.4 Testing

```
X_train, y_train = bcw_train.drop('target', axis=1), bcw_train['target']

mlp = MLPClassifier(hidden_layer_sizes=(20, 20), batch_size=50,
                    learning_rate_init=1e-3, learning_rate='constant',
                    shuffle=True, solver='adam', alpha=1e-2, random_state=42)
mlp.fit(X_train, y_train)
X_test, y_test = bcw_test.drop('target', axis=1), bcw_test['target']
y_pred = mlp.predict(X)
final_metrics = pd.DataFrame(data={'Accuracy': accuracy_score(y, y_pred),
                                   'Precision': accuracy_score(y, y_pred),
                                   'Recall': recall_score(y, y_pred),
                                   'F1-Score': f1_score(y, y_pred)},
                                   index=[0])
final_metrics['Model'] = 'MLP'

# Compute the confusion matrix
cm = confusion_matrix(y, y_pred)
# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

Listing 31: Code snippet for MLP testing

8.4.1 Final performance evaluation

	Accuracy	Precision	Recall	F1-Score	Model
0	1.000	1.000	1.000	1.000	MLP

Table 21: Performance metrics for MLP model

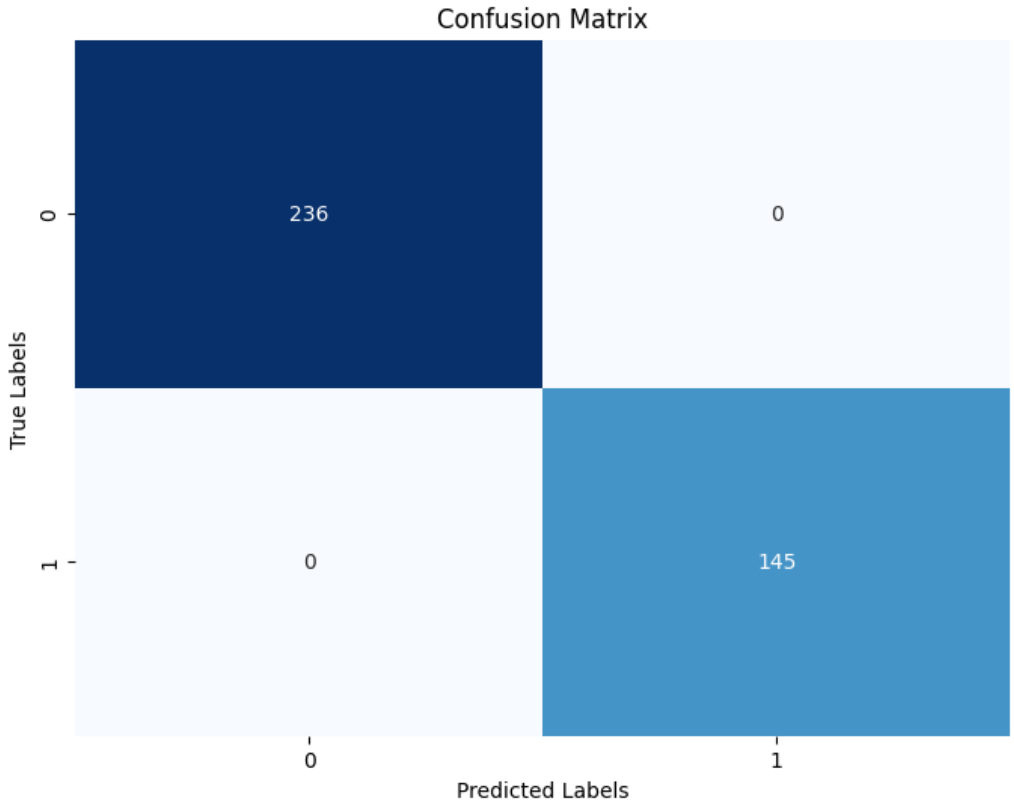

Figure 46: Confusion matrix for our MLP model.

The model's performance metrics, as displayed in 21, indicate exemplary results. With an accuracy, precision, recall, and F1-score all at a perfect 1.000, the model has successfully predicted every instance in the test data.

This is further evidenced by the confusion matrix shown in 46, where there are no false positives or false negatives; 236 samples are correctly classified as class 0 and 145 samples as class 1.

Achieving such results without overfitting is impressive and suggests that the model has generalized exceptionally well to the test data.

9 Discussion of Results

In the scope of this document, a range of SML models have been employed to make accurate predictions on the input data derived from the BCW dataset. The prevailing question that remains is to determine which model exhibits superior performance on this dataset.

To address this inquiry adequately, this discussion will be supplemented with relevant visual aids and datasets, facilitating an informed conclusion.

ROC curve for probabilistic models

Throughout this document I have programmed several probabilistic machine learning models, namely, the Support Vector Machine (SVM), Multilayer Perceptron (MLP), and Gaussian Naive Bayes.

In contrast, the Decision Tree and K-Nearest Neighbors (KNN) models are characterized as non-probabilistic. Their outputs are not probabilities but categorical model decisions.

Specifically, a Decision Tree assigns a sample to a particular class based on predefined criteria, whereas KNN classification depends on the majority vote of the 'K' nearest samples.

This inherent trait renders such models unsuitable for Receiver Operating Characteristic (ROC) curve analysis because their outputs are not contingent on a variable threshold value.

Observation of Figure 47 permits the conclusion that the SVM and MLP models demonstrate consistent and exceptionally high performance across varying threshold levels. The SVM is marginally superior, as evidenced by its AUC value of one, signifying its status as a perfect classifier. The MLP model, with an AUC of 0.99, is nearly equivalent to the SVM in performance, but marginally inferior. The Gaussian Naive Bayes (GNB) model also exhibits commendable performance, yet it does not reach the elevated benchmark set by the SVM and MLP models.

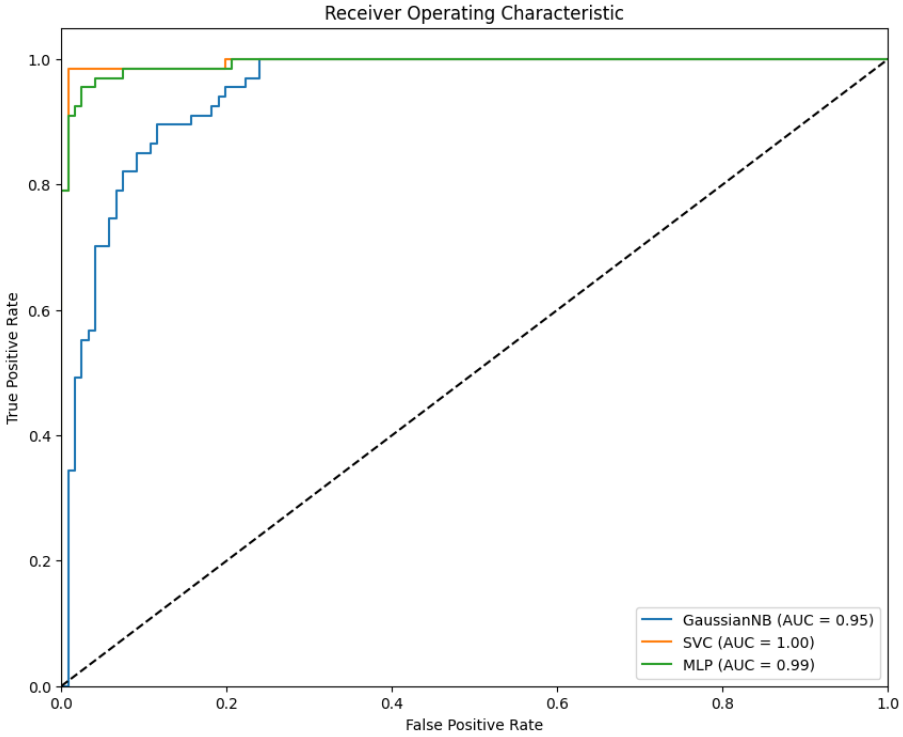

Figure 47: ROC Curves. This figure illustrates the ROC curve corresponding to each probabilistic machine learning model evaluated within this research. Additionally, the Area Under the Curve (AUC) metric for each model is depicted, providing a quantitative assessment of performance.

We may conclude that the SVM model outstrips others in performance, thus rendering it the most fitting SML model for the dataset in question. To substantiate this claim, visual depictions of the F1-Score and Accuracy for each SML model discussed in this document, Decision Tree and K-Nearest Neighbors, are also presented.
This graphic is also backed by the data depicted in Table 22.

Barplot for Performance Metrics

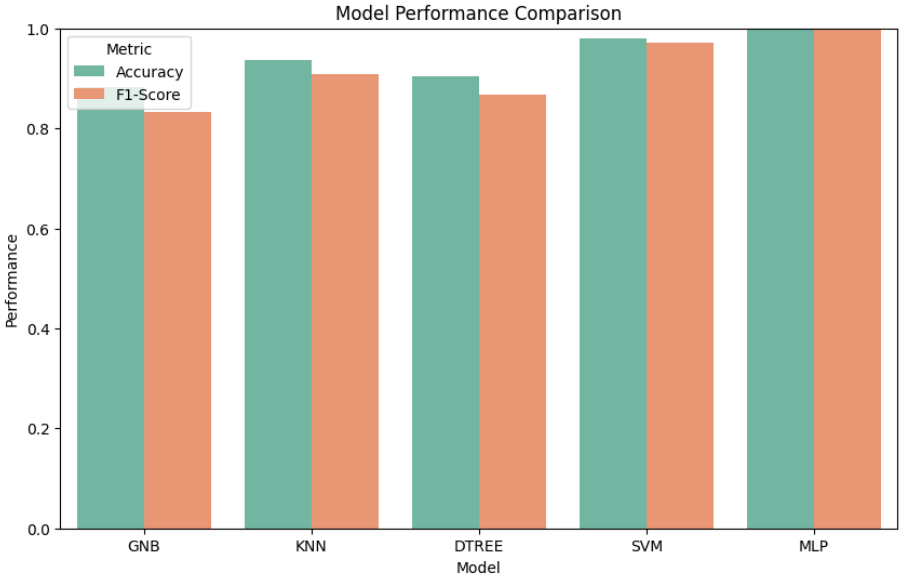

Figure 48: Accuracy and F1-Score Comparison Barplot. This barplot provides a comparative analysis of each model's performance using accuracy and F1-scores as evaluative metrics.

	Accuracy	Precision	Recall	F1-Score	Model
0	0.883	0.883	0.821	0.833	GNB
1	0.936	0.936	0.881	0.908	KNN
2	0.904	0.904	0.881	0.868	DTREE
3	0.979	0.979	0.985	0.971	SVM
4	1.000	1.000	1.000	1.000	MLP

Table 22: Performance Metrics for Supervised Machine Learning Models for a 0.5 threshold. This table shows the performance metrics for each supervised machine learning model covered in this document, including precision and recall values, complementing the barplot which illustrates accuracy and F1-Scores.

In conclusion, upon analysis of Figure 48 and Table 22, it becomes apparent that the MLP emerges as the optimal classifier, a finding that diverges from earlier inferences drawn from the ROC curve analysis in Figure 47.
Consequently, it is asserted that the MLP, as configured in Listing 31, is the most appropriate model for the prediction of data in the BCW dataset, following the preprocessing steps executed in this study.

Author's Note:

I would like to clarify that the code and all its associated content presented in this document are my own work and have been created for the purpose of fulfilling an academic homework assignment. This work is a result of my own understanding and efforts in the context of the assignment's requirements.

Source Code

The source code and additional resources for this project are available in my GitHub repository. You can access it through the following link: GitHub Repository.

Mario Pascual González

November 9, 2023

References

[1] W. W. M. O. S. Nick and S. W., "Breast Cancer Wisconsin (Diagnostic)," UCI Machine Learning Repository, 1995, DOI: https://doi.org/10.24432/C5DW2B.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[3] S. learn Developers. (2023) Decision tree classifier example with iris dataset. Accessed: 2023-10-02. [Online]. Available: https://scikit-learn.org/stable/auto_examples/tree/plot_iris_dtc.html#sphx-glr-auto-examples-tree-plot-iris-dtc-py

[4] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, "Classification and regression trees," *CRC press*, 1986.

[5] C. E. Shannon, *A mathematical theory of communication*. University of Illinois Press, 2001.

[6] T. Zhang, "Statistical behavior and consistency of classification methods based on convex risk minimization," *The Annals of Statistics*, pp. 56–134, 2004.

[7] N. Johnson, P. Vulimiri, A. To, X. Zhang, C. Brice, B. Kappes, and A. Stebner, "Machine learning for materials developments in metals additive manufacturing," 05 2020.

[8] Y. Yin, J. Jang-Jaccard, W. Xu, A. Singh, J. Zhu, F. Sabrina, and J. Kwak, "Igrf-rfe: a hybrid feature selection method for mlp-based network intrusion detection on unsw-nb15 dataset," *Journal of Big Data*, vol. 10, no. 1, p. 15, 2023. [Online]. Available: https://doi.org/10.1186/s40537-023-00694-8

[9] A. Amini, A. Soleimany, S. Karaman, and D. Rus, "Spatial uncertainty sampling for end-to-end control," 2019.