Zoidberg 2.0

Introduction

We were given a dataset of x-rays, which took around 20 minutes to install from their server. To speed things up and for convenience we sanitized the dataset and uploaded it to hugging face hub. We will see how fast and concise this solution turned out to be.

We outsourced the following data processing tasks to these files :

- Finding the right image mask
- Data preprocessing and saving in `.npz` file

Our goal with the dataset is to find the right machine learning model to help doctors in detecting pneumonia. We will proceed by pulling the `train` split of the dataset (as you can see with the `split="train"` parameter).

We will begin by loading the **preprocessed** dataset from the file.

```python
import numpy as np

processed_data = np.load("./datasets/processed_data.npz",
allow_pickle=True)

x_train = processed_data["x_train"]
y_train = processed_data["y_train"]
x_test = processed_data["x_test"]
y_test = processed_data["y_test"]
x_eval = processed_data["x_eval"]
y_eval = processed_data["y_eval"]

import os

if os.path.exists("./datasets/x_train_pca.npy"):
    x_train_pca = np.load("./datasets/x_train_pca.npy")
    x_test_pca = np.load("./datasets/x_test_pca.npy")
    x_eval_pca = np.load("./datasets/x_eval_pca.npy")
```

Next, we will calculate the offset from the geometric center of each image in the training set.

Calculating the Offset

Calculating the offset between the center of mass (COM) and the geometric center of each image is crucial for several reasons:

1. **Alignment and Normalization**:
    - By understanding the offset, we can align the images more accurately. This alignment helps in normalizing the dataset, ensuring that the features of interest

(e.g., lungs in x-ray images) are consistently positioned across all images. This consistency is vital for the performance of machine learning models.

2. **Feature Extraction**:
   - The offset can serve as an additional feature for machine learning models. For instance, the displacement of the COM might reveal patterns or anomalies specific to certain conditions, such as pneumonia. Including this feature can enhance the model's ability to distinguish between different classes.

Now let's take a look at how we can proceed with calculating the offset.

Function: `get_center_of_mass`

This function calculates the center of mass (COM) of a binary image.

- **Purpose**:
  - To compute the average position of all the foreground pixels in a binary image.
- **Parameters**:
  - `image` (numpy.ndarray): The input binary image represented as a NumPy array.
- **Returns**:
  - `center` (tuple[int, int]): A tuple containing the coordinates (row, column) of the center of mass of the binary image. If the center of mass cannot be computed (e.g., if the image is empty), it returns `(0, 0)`.
- **Steps**:
  a. **Convert to Binary Image**:
     - The input image is converted to a binary image where pixels with values greater than 128 are considered foreground (`True` or `1`), and others are considered background (`False` or `0`).

  b. **Compute Center of Mass**:
     - The `center_of_mass` function from the `scipy.ndimage` module is used to compute the center of mass of the binary image.

  c. **Handle NaN Values**:
     - If the computed center of mass contains any `NaN` values, the function returns `(0, 0)`.
     - Otherwise, it returns the computed center of mass.
- **Example**:
  - If the input image is a binary image with foreground pixels forming a shape, the function will return the average position of those pixels.

```python
from scipy.ndimage import center_of_mass

def get_center_of_mass(image: np.ndarray) -> tuple[int, int]:
    binary_image = np.array(image) > 128
    com = center_of_mass(binary_image)
    return (0, 0) if np.isnan(com).any() else com
```

Function: `calculate_offset`

This function calculates the offset between the center of mass (COM) and the geometric center of a binary image.

- **Purpose**:
  - To compute the displacement of the center of mass relative to the geometric center of the image.
- **Parameters**:
  - `image` (numpy.ndarray): The input binary image represented as a NumPy array.
- **Returns**:
  - `offset` (numpy.ndarray): A NumPy array representing the offset between the center of mass and the geometric center. The offset has the same dimensionality as the input image.
- **Steps**:
  a. **Compute Center of Mass**:
     - The function `get_center_of_mass` is called to compute the center of mass of the binary image.
  b. **Convert Image to NumPy Array**:
     - The input image is converted to a NumPy array.
  c. **Compute Geometric Center**:
     - The geometric center is computed as the center of the image grid, which is assumed to be at half the width and half the height of the image.
  d. **Calculate Offset**:
     - The offset is calculated as the difference between the geometric center and the center of mass.
  e. **Check for NaN Values**:
     - If any NaN values are found in the offset, an exception is raised.
  f. **Return Offset**:
     - The computed offset is returned.
- **Example**:
  - If the input image is a binary image with a certain shape, the function will return the displacement of the center of mass of that shape relative to the geometric center of the image.

```python
import math

def calculate_offset(image: np.ndarray) -> np.ndarray:
    com = get_center_of_mass(image)

    image = np.array(image)

    geometric_center = np.array(image.shape) / 2

    offset = geometric_center - com

    # Check for NaN in the tuple
    if any(math.isnan(x) for x in offset):
        raise Exception("nan found")

    return offset
```

Let's compute the offsets for each image in the training set.

```
offsets = [calculate_offset(img) for img in x_train]
```

Plotting Offsets of Center of Mass

This following step visualizes the offsets of the center of mass for images in the training set, distinguishing between normal and pneumonia cases.

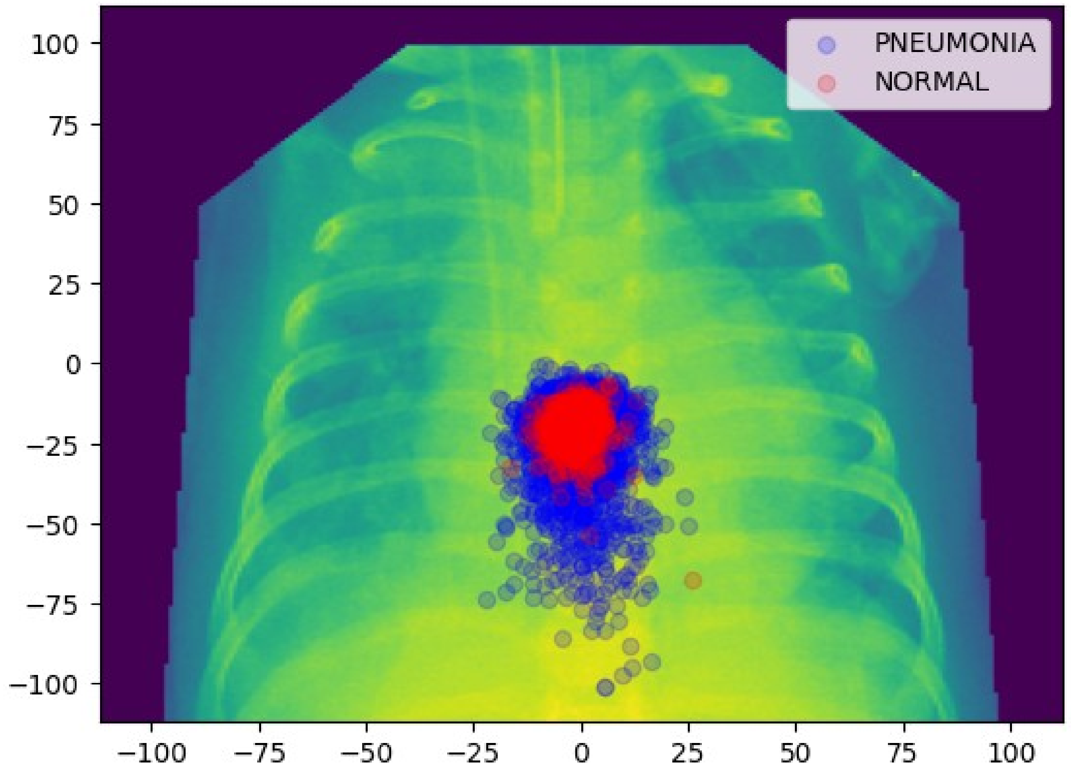
Steps:

1. **Extract Offsets**:
   - `x_offsets` and `y_offsets` are extracted from the `offsets` list, which contains the offsets of the center of mass for each image.
   - These offsets are then separated into `x_offsets_normal`, `x_offsets_pneumonia`, `y_offsets_normal`, and `y_offsets_pneumonia` based on the labels (`y_train`).
2. **Calculate Mean Offsets**:
   - The mean values of `x_offsets` and `y_offsets` are calculated and printed.
3. **Define Extent**:
   - The `extent` variable is defined to set the limits for the image display, assuming the images are 224x224 pixels.
4. **Display Image and Scatter Plot**:
   - A random image from the training set is displayed using `plt.imshow`.
   - Scatter plots of the offsets are overlaid on the image:
     - Blue points represent pneumonia cases.
     - Red points represent normal cases.

This visualization helps in understanding the distribution of the center of mass offsets for normal and pneumonia cases. It can reveal patterns such as whether the center of mass is more centralized in normal cases compared to pneumonia cases.

```
from matplotlib import pyplot as plt
import random

x_offsets = np.array([offset[1] for offset in offsets])
y_offsets = np.array([offset[0] for offset in offsets])
x_offsets_normal = x_offsets[y_train == 0]
x_offsets_pneumonia = x_offsets[y_train == 1]
y_offsets_normal = y_offsets[y_train == 0]
y_offsets_pneumonia = y_offsets[y_train == 1]

print("mean x offset : ", np.mean(x_offsets))
print("mean y offset : ", np.mean(y_offsets))

# Dividing by image sizes
extent = [-224 / 2, 224 / 2, -224 / 2, 224 / 2]

# Display the image with the calculated extents
plt.imshow(x_train[int(random.random() * len(x_train))],
extent=extent, aspect="auto")
plt.scatter(
    x_offsets_pneumonia, y_offsets_pneumonia, color="blue", alpha=0.2,
label="PNEUMONIA"
)
plt.scatter(x_offsets_normal, y_offsets_normal, color="red",
alpha=0.2, label="NORMAL")

plt.legend()
plt.show()
```

```
mean x offset :  -0.8275138212987382
mean y offset :  -20.179250145815896
```



We can observe that on the y-axis, the points are pulled towards the bottom. This can be explained by the presence of the visible **pelvis bone** in some of the images, which results in a downward pull of the **center of mass**.

Additionally, it is interesting to note that the images of `NORMAL` patients exhibit a more grouped and centralized **center of mass** compared to those of patients with `PNEUMONIA`.

Dimension Reduction

We will first do a **Principal Component Analysis** (`PCA`) to reduce the dimensionality of the images, and then we will plot the first two `PCA`s and then the third in a 3-D plot.

Why PCA?

Overview

Principal Component Analysis (PCA) is a statistical technique used for dimensionality reduction while preserving as much variance as possible in the dataset. In the context of this notebook, PCA is applied to the image data to achieve several key objectives.

Reasons for Performing PCA

1. **Dimensionality Reduction**:
   - **High-Dimensional Data**: Image data typically has a high number of dimensions (e.g., a 224x224 image has 50,176 dimensions). High-dimensional data can be computationally expensive to process and analyze.
   - **Reducing Complexity**: PCA reduces the number of dimensions by transforming the data into a new set of orthogonal components (principal components) that capture the most variance in the data. This makes the data more manageable and reduces computational complexity.
2. **Noise Reduction**:
   - **Filtering Noise**: By focusing on the principal components that capture the most variance, PCA helps in filtering out noise and less important features. This can lead to cleaner data and improved performance of machine learning models.
3. **Visualization**:
   - **2D and 3D Plots**: PCA allows for the visualization of high-dimensional data in 2D or 3D plots. This can help in understanding the distribution and relationships within the data.
   - **Pattern Recognition**: Visualizing the principal components can reveal patterns, clusters, and separations between different classes (e.g., `NORMAL` vs. `PNEUMONIA` cases).

Function: `normalize_images`

This function takes a list of images and processes them by flattening each image into a 1D vector and normalizing the pixel values to be between 0 and 1.

Parameters:

- `images` (list or np.ndarray): A list or array of images to be normalized. Each image is expected to be in a format that can be converted to a NumPy array.

Returns:

- `np.ndarray`: A NumPy array where each image has been flattened into a 1D vector and its pixel values normalized to the range [0, 1].

```
def normalize_images(images: np.ndarray) -> np.ndarray:
    return np.array([np.asarray(img).ravel() / 255.0 for img in
images])
```

Let's first try keeping all components for PCA without limiting `n_components` :

```
from sklearn.decomposition import PCA

pca = PCA()

x_train_pca_full = pca.fit_transform(normalize_images(x_train))
x_test_pca_full = pca.transform(normalize_images(x_test))
x_eval_pca_full = pca.transform(normalize_images(x_eval))
```

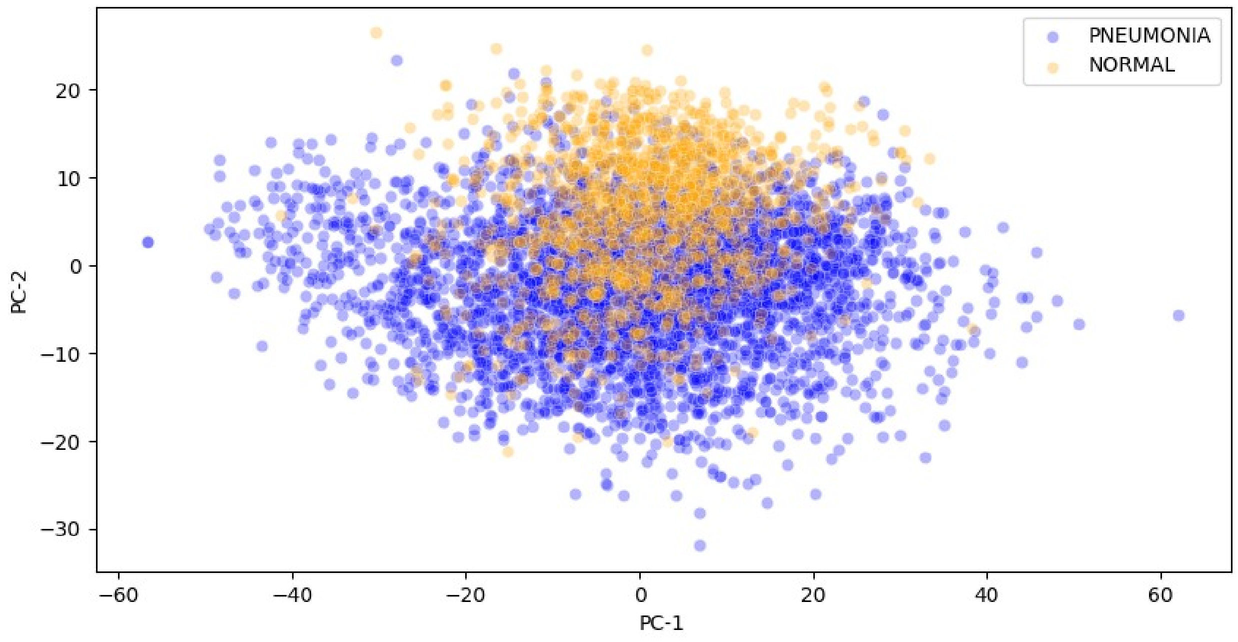
Exploring the PCA Components

In this section, we visualize the first two and first three principal components obtained from the PCA transformation of the training data. This scatter plot helps in understanding the distribution and separation of the data points for the two classes: `PNEUMONIA` and `NORMAL`.

Principal Components 1 and 2:

- We use `seaborn.scatterplot` to plot the first two principal components for the `PNEUMONIA` cases. The points are colored blue.
- Similarly, we plot the first two principal components for the `NORMAL` cases. The points are colored orange
- We label the x-axis as "PC-1" and the y-axis as "PC-2" to indicate the first and second principal components, respectively.

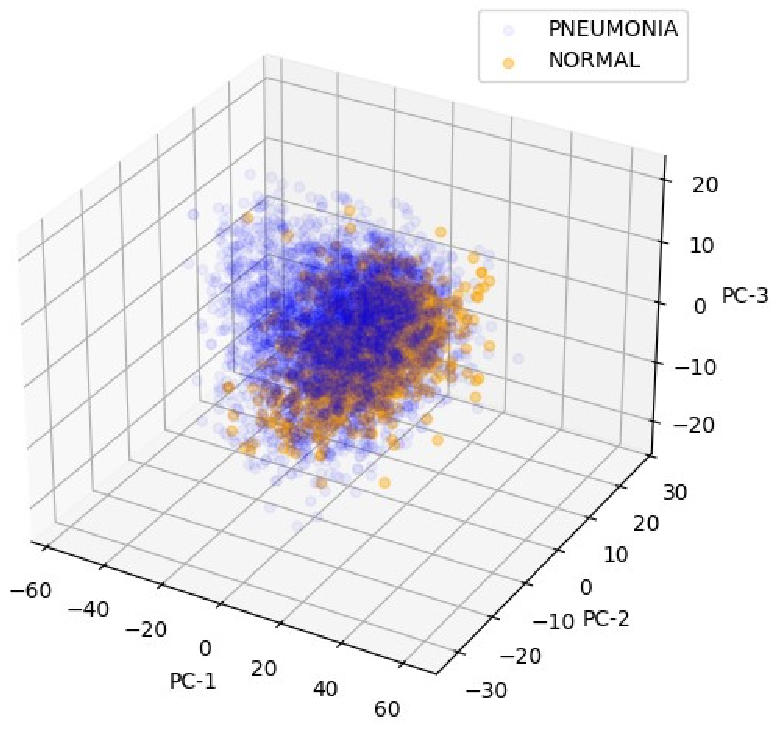
```
%matplotlib inline
import seaborn as sns

plt.figure(figsize=(10, 5))
sns.scatterplot(x=x_train_pca[y_train == 1, 0], y=x_train_pca[y_train
== 1, 1], color="blue", label="PNEUMONIA", alpha=0.3)
sns.scatterplot(x=x_train_pca[y_train == 0, 0], y=x_train_pca[y_train
== 0, 1], color="orange", label="NORMAL", alpha=0.3)
plt.xlabel("PC-1"), plt.ylabel("PC-2")
plt.show()
```



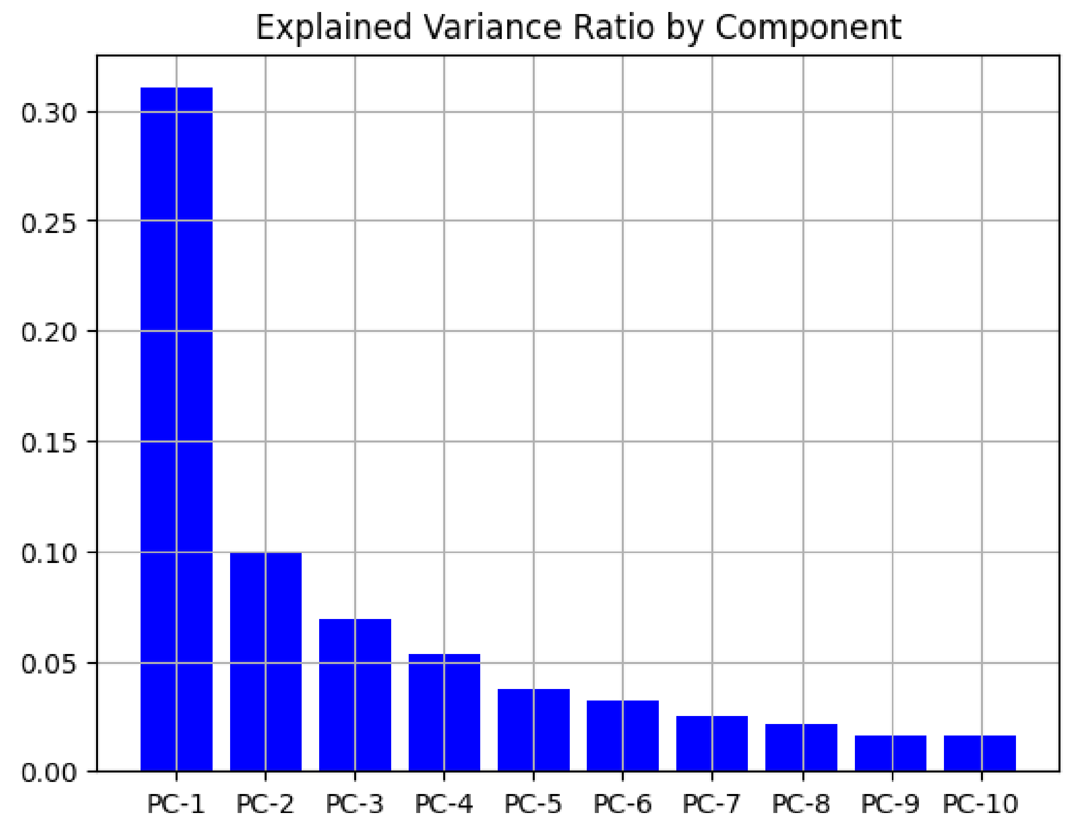
Principal Components 1, 2 and 3:

This time we will use a 3D interactive plot.

```
%matplotlib widget

# Assuming X_3d is your 3D data array and labels is your corresponding
labels array
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection="3d")

ax.scatter(
    x_train_pca[y_train == 1, 0],
    x_train_pca[y_train == 1, 1],
    x_train_pca[y_train == 1, 2],
    color="blue",
    label="PNEUMONIA",
    alpha=0.05
)

ax.scatter(
    x_train_pca[y_train == 0, 0],
    x_train_pca[y_train == 0, 1],
    x_train_pca[y_train == 0, 2],
    color="orange",
    label="NORMAL",
    alpha=0.4
)

ax.set_xlabel('PC-1')
ax.set_ylabel('PC-2')
ax.set_zlabel('PC-3')

plt.legend()
plt.show()
```



```
%matplotlib inline

explained_variance_ratio_10 = pca.explained_variance_ratio_[:10]
pc_10 = [f"PC-{i}" for i in range(1, len(explained_variance_ratio_10)
+ 1)]

plt.bar(pc_10, explained_variance_ratio_10, color="blue")
plt.title("Explained Variance Ratio by Component")
plt.grid(True)
plt.show()
```



```
x_test_pca = pca.transform(normalize_images(x_test))
x_eval_pca = pca.transform(normalize_images(x_eval))
```

Function: `components_for_variance_percentage`

This function calculates the number of principal components required to achieve a specified percentage of explained variance in a dataset.

Parameters:

- `variance_ratios` (numpy.ndarray):
  - An array containing the explained variance ratios for each principal component.
- `per` (float, optional, default=0.99):
  - The desired percentage of variance to be explained, specified as a float in the range [0, 1].

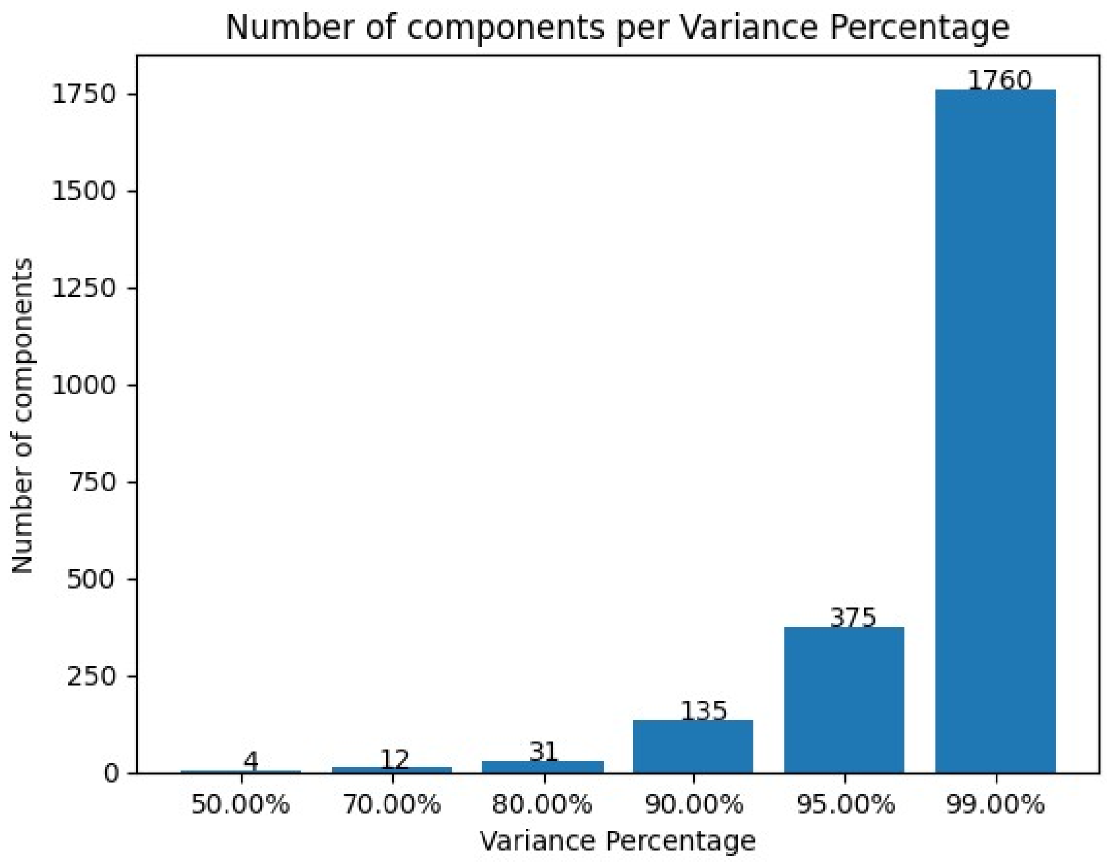
Returns:

- `int`:
  - The number of principal components needed to explain the specified percentage of variance.

```
def components_for_variance_percentage(variance_ratios: np.ndarray,
per: float = 0.99):
    cumulative_variance = np.cumsum(variance_ratios)
    return np.argmax(cumulative_variance >= per) + 1
```

By using a bar plot, we can visualize the number of components needed for each variance percentage :

```
%matplotlib inline

percentages = [0.5, 0.7, 0.8, 0.9, 0.95, 0.99]
num_components = [
    components_for_variance_percentage(pca.explained_variance_ratio_,
per)
    for per in percentages
]

fig, ax = plt.subplots()
bars = ax.bar(["{:.2%}".format(value) for value in percentages],
num_components)
for bar in bars:
    yval = bar.get_height()
    ax.text(
        (bar.get_x() + (bar.get_width() / 2) -
(math.floor(len(str(yval)) / 2) / 10)),
        yval + 0.005,
        yval,
    )

ax.set_title("Number of components per Variance Percentage")
ax.set_xlabel("Variance Percentage")
ax.set_ylabel("Number of components")
plt.show()
```

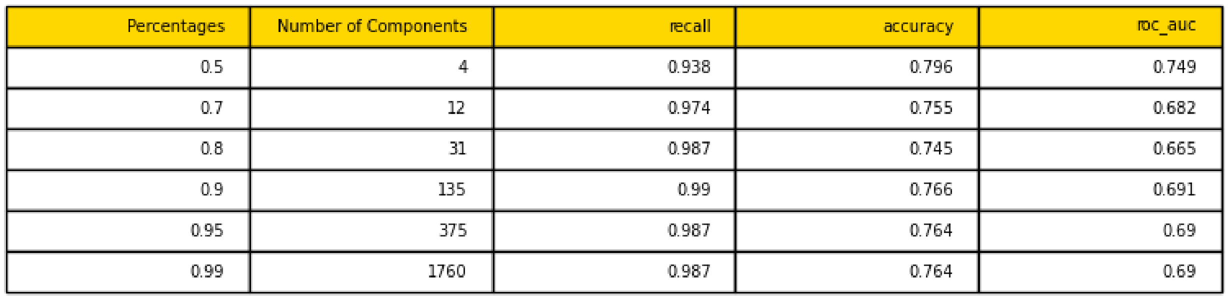


Finding the Optimal Number of PCA Components

In this section, we aim to determine the optimal number of Principal Component Analysis (PCA) components for our machine learning models. By evaluating the performance of an `SVC` model and a `RandomForestClassifier` across different intervals of explained variance, we can identify the most effective dimensionality reduction strategy.

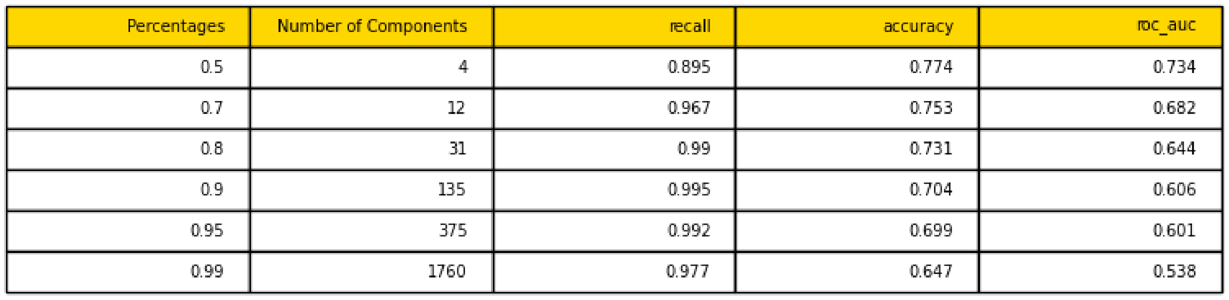
NOTE: For simplicity, these two models are evaluated without hyperparameter tuning. We will do it in the next section.

```
percentage_to_ncomponents = {
    percentages[i]: num_components[i] for i in
range(len(num_components))
}

from sklearn.metrics import accuracy_score, recall_score,
roc_auc_score

def evaluate_model_with_pca_components(model):
    model_scores = []

    for _, n_components in percentage_to_ncomponents.items():
        model.fit(x_train_pca_full[:, :n_components], y_train)
        y_pred = model.predict(x_test_pca_full[:, :n_components])

        recall = recall_score(y_test, y_pred)
        accuracy = accuracy_score(y_test, y_pred)
        roc_auc = roc_auc_score(y_test, y_pred)

        model_scores.append([round(score, 3) for score in [recall,
accuracy, roc_auc]])

    return model_scores

def plot_model_scores(model_scores):
    scoring = ["recall", "accuracy", "roc_auc"]

    svc_scores_data = [
        [percentage, n_components, *scores]
        for percentage, n_components, scores in zip(
            list(percentage_to_ncomponents.keys()),
            list(percentage_to_ncomponents.values()),
            model_scores
        )
    ]

    # Add table heading
    svc_scores_data.insert(0, ["Percentages", "Number of Components"]
+ scoring)

    _, ax = plt.subplots()
    table = ax.table(cellText=svc_scores_data, loc="center")
    table.set_fontsize(12)
    table.scale(1.5, 1.5)

    for i in range(len(svc_scores_data[0])):
        table[(0, i)].set_facecolor("#FFD700")
    ax.axis("off")
    plt.show()

from sklearn.svm import SVC

svc = SVC()
svc_scores = evaluate_model_with_pca_components(svc)
plot_model_scores(svc_scores)
```



Percentages	Number of Components	recall	accuracy	roc_auc
0.5	4	0.938	0.796	0.749
0.7	12	0.974	0.755	0.682
0.8	31	0.987	0.745	0.665
0.9	135	0.99	0.766	0.691
0.95	375	0.987	0.764	0.69
0.99	1760	0.987	0.764	0.69

```python
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_jobs=-1)
clf_scores = evaluate_model_with_pca_components(clf)
plot_model_scores(clf_scores)
```



Percentages	Number of Components	recall	accuracy	roc_auc
0.5	4	0.895	0.774	0.734
0.7	12	0.967	0.753	0.682
0.8	31	0.99	0.731	0.644
0.9	135	0.995	0.704	0.606
0.95	375	0.992	0.699	0.601
0.99	1760	0.977	0.647	0.538

After evaluating these two models, we can conclude that although `n_components=4` achieves the highest `accuracy` and `roc_auc` scores, it performs poorly in `recall`, resulting in a high number of `False Negatives`, which is undesirable in healthcare applications.

This poor performance in recall may be due to retaining only 50% of the variance.

On the other hand, `n_components=31` offers a good balance across all metrics while maintaining a relatively low number of components. Therefore, we will proceed with `n_components=31` moving forward.

```python
from sklearn.decomposition import PCA

pca = PCA(n_components=31)

x_train_pca = pca.fit_transform(normalize_images(x_train))

x_test_pca = pca.transform(normalize_images(x_test))
x_eval_pca = pca.transform(normalize_images(x_eval))

np.save("./datasets/x_train_pca", x_train_pca)
np.save("./datasets/x_test_pca", x_test_pca)
np.save("./datasets/x_eval_pca", x_eval_pca)
```

Model Discovery

We will use `RandomizedSearchCV` with the `recall` scoring to be able to test a wide array of hyperparameters in the most efficient way possible.

The reason why we use `recall` scoring is that in healthcare applications, it is crucial to minimize the number of false negatives. Recall measures the ability of the model to correctly identify all relevant instances (i.e., all actual positive cases). A high recall score ensures that most of the positive cases (e.g., pneumonia cases) are correctly identified, which is critical in medical diagnostics. We will see an example of how it works for `SVC` and for the rest we'll just use the score.

In addition, for each model, after extracting the best parameters we will `train` and `evaluate` them with the found parameters using **Cross Validation** (with `StratifiedKFold` to preserve percentage of samples of each class) whilst keeping track of `["recall", "accuracy", "roc_auc"]` scores for maximum generalization.

However, before starting with fine-tuning and testing models, we will scale the values in the goal of reducing the effect outliers will have on the models.

We will use the `MaxAbsScaler` because it does not shift/center the data and does not destroy any sparsity.

```python
from sklearn.preprocessing import MaxAbsScaler
```

The reason why we're using `RandomizedGridSearchCV` with 60 iterations instead of `GridSearch` is because of this amazing research paper by *Bergstra & Bengio*, *Random Search for Hyper-Parameter Optimization*. In which they showed that, in surprisingly many instances, random search performs about as well as grid search. All in all, trying 60 random points sampled from the grid seems to be good enough.

Support Vector Classification

```
# Speeding execution time using Intel(R) Extension for Scikit-learn
from sklearnex import patch_sklearn

patch_sklearn()
```

```
Intel(R) Extension for Scikit-learn* enabled
(https://github.com/intel/scikit-learn-intelex)
```

```
from sklearn.model_selection import RandomizedSearchCV,
StratifiedKFold

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

def randomized_search_cv(estimator, param_distributions: dict):
    rand_search_cv = RandomizedSearchCV(
        estimator,
        param_distributions,
        n_iter=60,
        scoring="roc_auc",
        cv=cv,
        random_state=42,
    )
    scaled_train_pca = MaxAbsScaler().fit_transform(x_train_pca)
    rand_search_cv_results = rand_search_cv.fit(scaled_train_pca,
y_train)
    return rand_search_cv_results

from sklearn.svm import SVC

svc = SVC()

param_distributions = {
    "C": [0.1, 1, 10, 100, 1000],
    "kernel": ["rbf", "poly", "sigmoid"],
    "degree": list(range(1, 11)),  # Just for "poly"
}

svc_search = randomized_search_cv(svc, param_distributions)
svc_search.best_params_
```

```
{'kernel': 'rbf', 'degree': 3, 'C': 1}
```

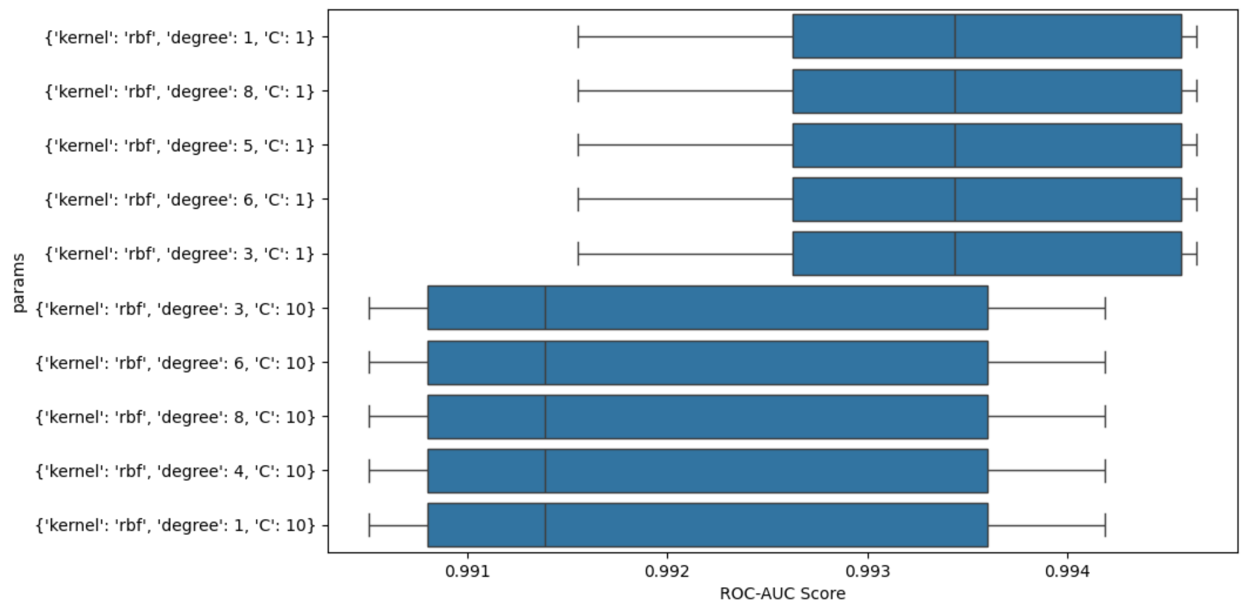
Function: `plot_cv_results`

This function visualizes the cross-validation results from a hyperparameter search using a box plot.

Parameters:

- `cv_results_` (dict): The cross-validation results from a hyperparameter search obtained from `RandomizedSearchCV`. This dictionary contains the scores and parameters for each fold and combination.
- `title` (str): The title of the plot. Default is an empty string.

```
import re
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

def plot_cv_results(cv_results_: dict, title: str="") -> None:
    df = pd.DataFrame(cv_results_)
    df = df.sort_values(by="rank_test_score")
    df = df.head(10)

    pattern = re.compile(r"split[0-9]+_test_score")

    split_columns = [c for c in df.columns.values.tolist() if 
pattern.match(c)]

    # Picking only the columns to plot
    df = df[["params"] + split_columns]
    df["params"] = df["params"].astype(str)

    # Rotating the dataframe
    df = df.melt(id_vars="params", value_vars=split_columns)

    plt.figure(figsize=(10, 6))
    sns.boxplot(df, x="value", y="params", fliersize=0)
    plt.xlabel("ROC-AUC Score")
    plt.title(title)
    plt.show()

%matplotlib inline

plot_cv_results(svc_search.cv_results_)
```



Let's store the models scores in a variable for future performance comparison:

```
models_scores = {}
```

Function: `evaluate_model`

This function evaluates a machine learning model using the best hyperparameters found during cross-validation.

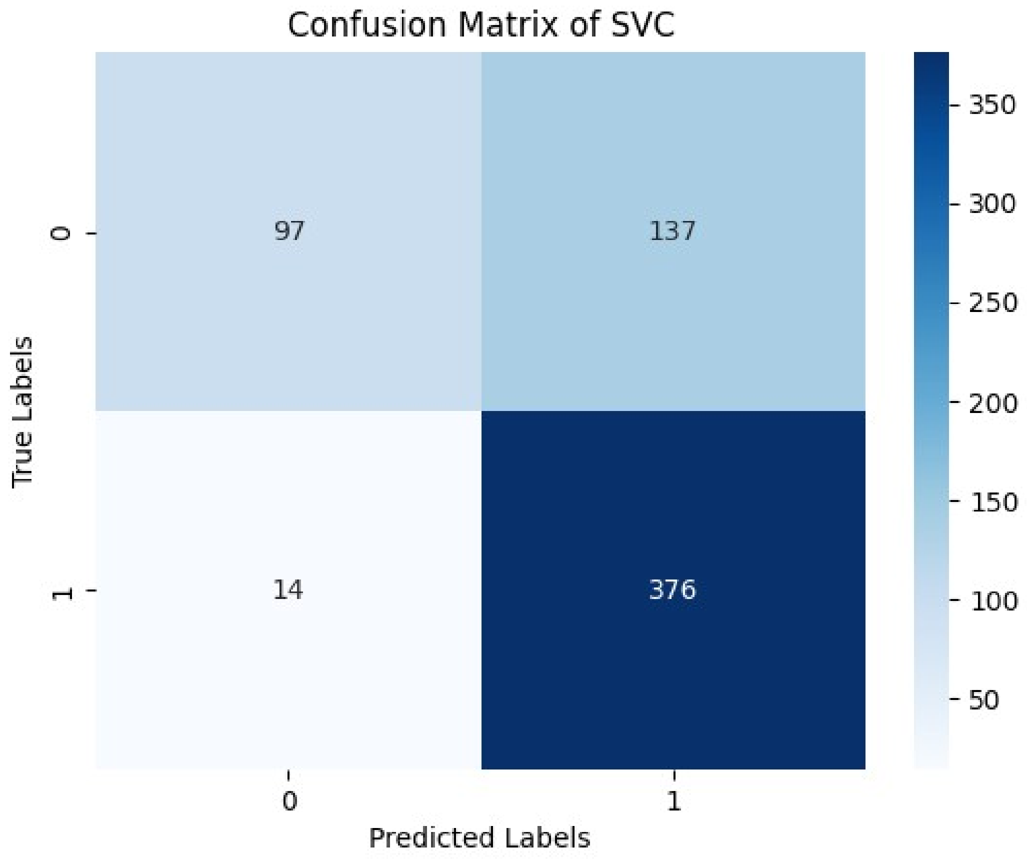
Purpose:

- To set the model's parameters to the best combination found during cross-validation.
- To train the model with these parameters.
- To evaluate the model's performance on the test set.
- To plot the confusion matrix for the model's predictions.

Parameters:

- `model` (sklearn.base.BaseEstimator): The machine learning model to be evaluated.
- `cv` (sklearn.model_selection._search.BaseSearchCV): The cross-validation search object containing the best parameters.

```
from sklearn.metrics import (
    confusion_matrix,
    accuracy_score,
    roc_auc_score,
    recall_score,
)
from src.utils.helpers import plot_confusion_matrix

def evaluate_model(model, cv):
    # set the params to be the best combination
    model_name = type(model).__name__
    model.set_params(**cv.best_params_)

    scaled_x_train_pca = MaxAbsScaler().fit_transform(x_train_pca)
    scaled_x_test_pca = MaxAbsScaler().fit_transform(x_test_pca)

    # train the model with the best params combination
    model.fit(scaled_x_train_pca, y_train)

    y_pred = model.predict(scaled_x_test_pca)

    models_scores[model_name] = {
        "recall": recall_score(y_test, y_pred),
        "accuracy": accuracy_score(y_test, y_pred),
        "roc_auc": roc_auc_score(y_test, y_pred)
    }

    cm = confusion_matrix(y_test, y_pred)
    plot_confusion_matrix(cm, model_name)

evaluate_model(svc, svc_search)
```

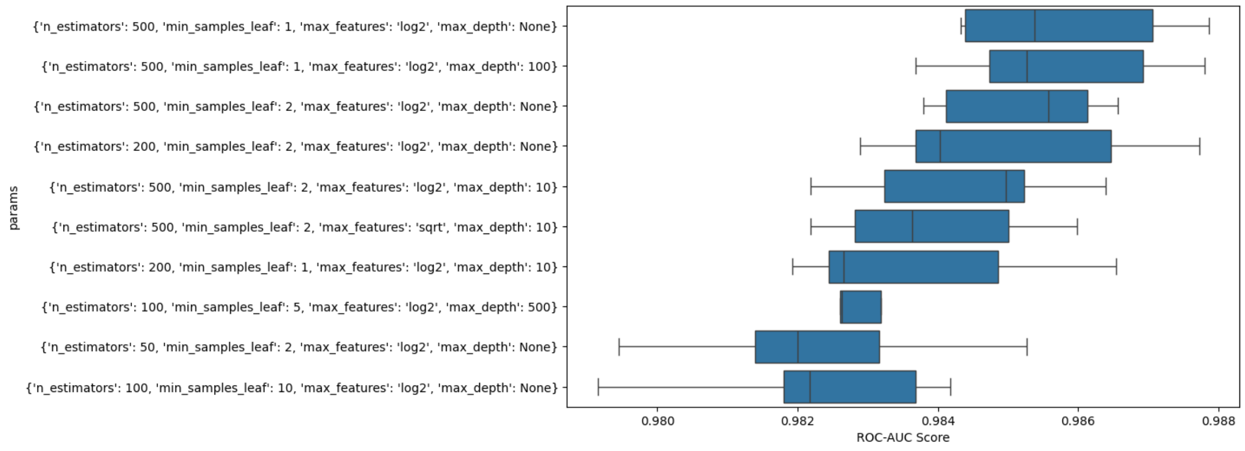


RandomForestClassifier

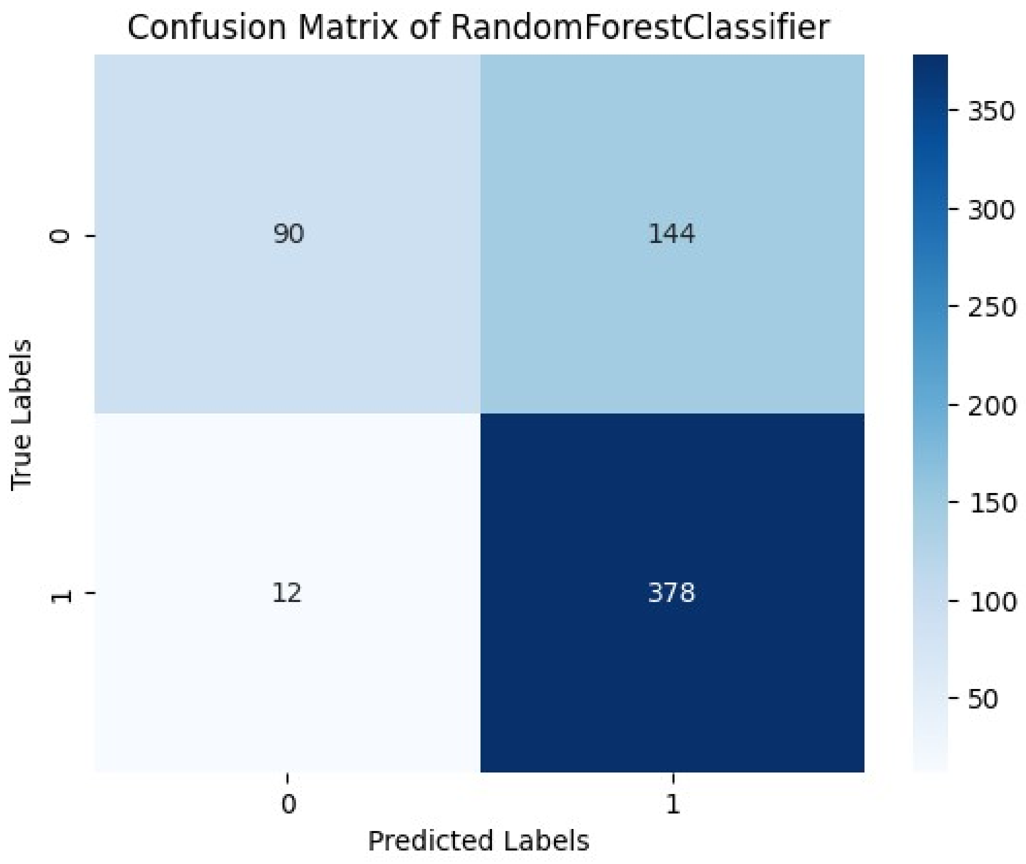
```
from sklearn.ensemble import RandomForestClassifier

param_distributions = {
    "n_estimators": [10, 50, 100, 200, 500],
    "max_depth": [10, 100, 500, 1000, None],
    "max_features": ["sqrt", "log2", 0.5],
    "min_samples_leaf": [1, 2, 5, 10, 50, 100]
}

rfc = RandomForestClassifier(n_jobs=-1)

rfc_search = randomized_search_cv(rfc, param_distributions)
rfc_search.best_params_
```

```
{'n_estimators': 500,
 'min_samples_leaf': 1,
 'max_features': 'log2',
 'max_depth': None}
```

```
%matplotlib inline

plot_cv_results(rfc_search.cv_results_)
```

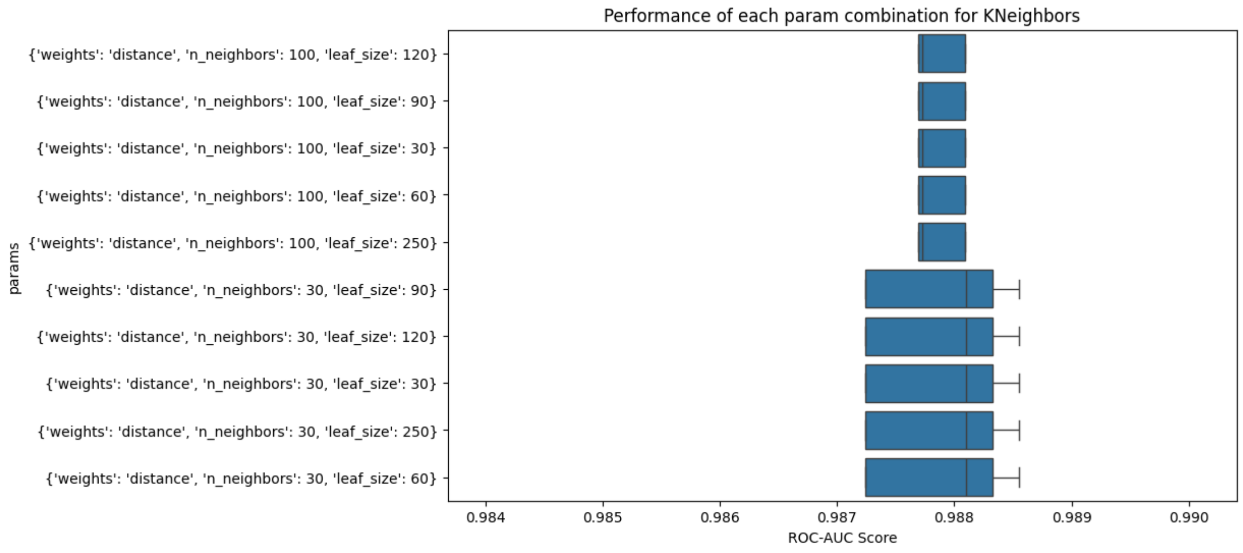


```
evaluate_model(rfc, rfc_search)
```

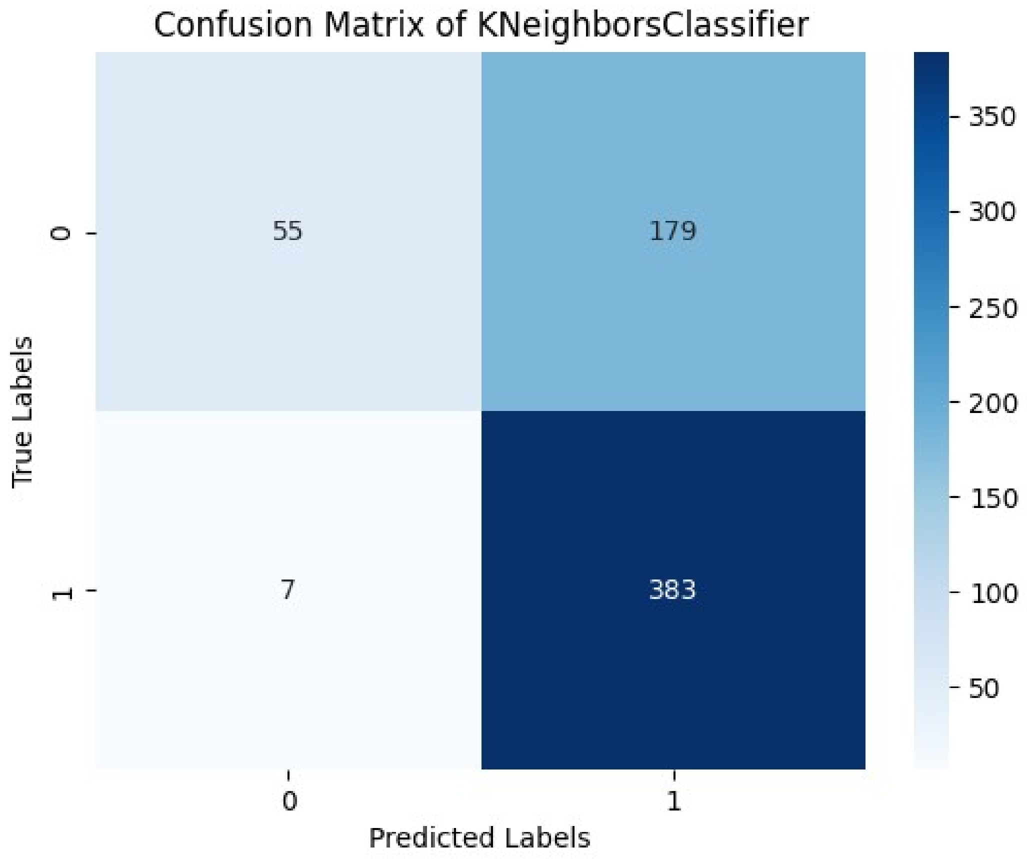


KNeighborsClassifier

```
from sklearn.neighbors import KNeighborsClassifier

param_distributions = {
    "n_neighbors": [10, 20, 30, 100, 250, 500],
    "weights": ["uniform", "distance"],
    "leaf_size": [30, 60, 90, 120, 250],
}

knn = KNeighborsClassifier(n_jobs=-1)
knn_search = randomized_search_cv(knn, param_distributions)
knn_search.best_params_

{'weights': 'distance', 'n_neighbors': 100, 'leaf_size': 30}

%matplotlib inline

plot_cv_results(knn_search.cv_results_, "Performance of each param
combination for KNeighbors")
```

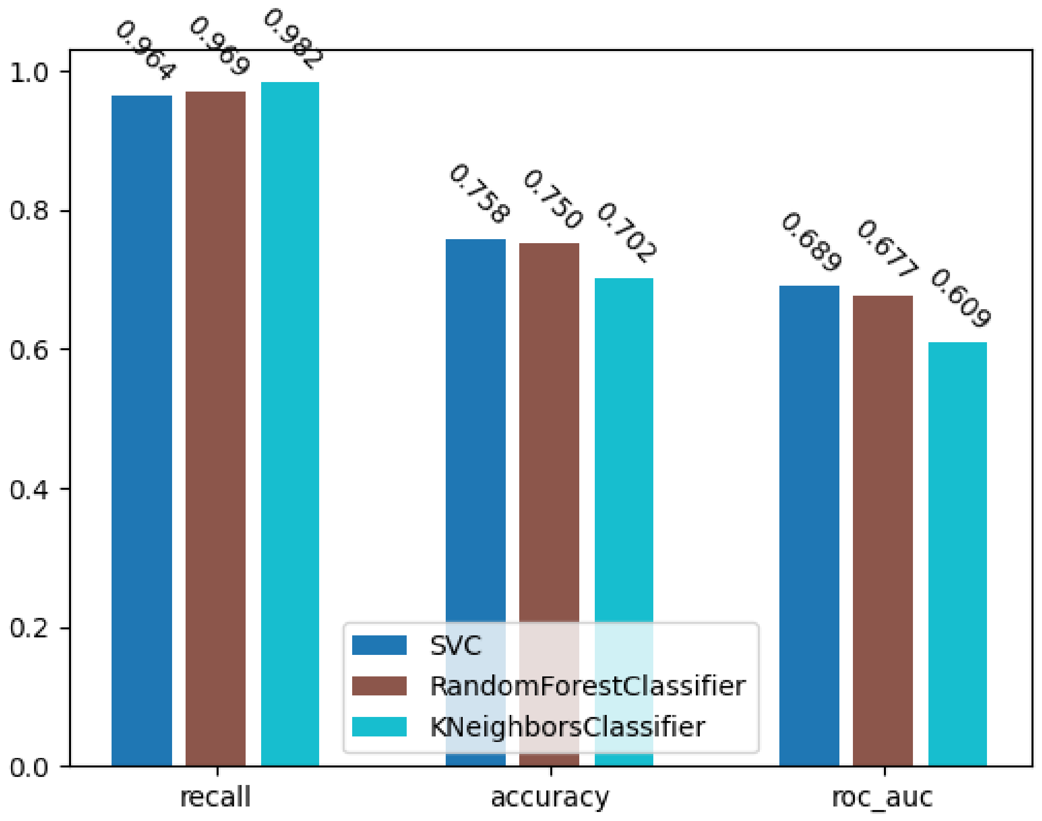


```
evaluate_model(knn, knn_search)
```



```
from matplotlib import pyplot as plt
from src.utils.helpers import generate_values_around_median
```

```python
def plot_models_scores(model_scores):
    average_model_scores = {
        model: {metric: np.mean(scores) for metric, scores in
metrics.items()}
        for model, metrics in model_scores.items()
    }

    metrics = list(next(iter(average_model_scores.values())).keys())

    # Get the number of models
    num_models = len(average_model_scores)
    num_metrics = len(metrics)

    # Generate a list of colors
    colors = plt.cm.tab10(np.linspace(0, 1, num_models))

    fig, ax = plt.subplots()
    index = np.arange(num_metrics) * num_models * 1.5

    for j, metric in enumerate(metrics):
        medians = generate_values_around_median(index[j], num_models)
        for i, (model, scores) in
enumerate(average_model_scores.items()):
            ax.bar(medians[i], scores[metric], color=colors[i])
            ax.text(
                medians[i],
                scores[metric] + 0.01,
                f"{scores[metric]:.3f}",
                ha="center",
                va="bottom",
                rotation=-45
            )

    ax.set_xticks(index)
    ax.set_xticklabels(metrics)
    plt.legend(list(average_model_scores.keys()), loc="lower center")
    plt.show()

plot_models_scores(models_scores)
```


0.964
0.969
0.982
0.758
0.750
0.702
0.689
0.677
0.609
1.0
0.8
0.6
0.4
0.2
0.0
SVC
RandomForestClassifier
KNeighborsClassifier
recall
accuracy
roc_auc