Machine Learning Project

Project Proposal

Team 5

Name	Sec	BN	ID	Email
بموا عريان عياد	1	17	9202391	bemoi.tawadros00@eng-st.cu.edu.eg
مارك ياسر نبيل	2	14	9203106	mark.ibrahim00@eng-st.cu.edu.eg
بيتر عاطف فتحي	1	18	9202395	peter.zaki00@eng-st.cu.edu.eg
كريم محمود كمال	2	12	9203076	kairm.mohamed003@eng-st.cu.edu.eg

Agenda

iii. Learning Curves Plot
iv. Hyperparameter Tuning
    - Grid Search
    - Train-Validation Curve
v. Bias-variance Analysis

c. Support Vector Machines
i. Feature Importance Plot
ii. Learning Curves Plot
iii. Partial Dependence Plot
iv. Hyperparameter Tuning
    - Grid Search
    - Train-Validation Curve
v. Bias-variance Analysis

d. Random Forest
i. Feature Importance Plot
ii. Learning Curves Plot
iii. Partial Dependence Plot
iv. Hyperparameter Tuning
    - Number of estimators effect
    - Grid Search
    - Train-Validation Curve
v. Bias-variance Analysis
vi. Tree Plot
    - Variable importance
    - Interactions between features
    - Overfitting

e. Adaboost
i. Feature Importance Plot
ii. Learning Curves Plot
iii. Partial Dependence Plot
iv. Hyperparameter Tuning
    - Number of estimators effect
    - Grid Search
    - Train-Validation Curve
v. Bias-variance Analysis

Santander Customer Transaction Prediction

Problem Definition:

The problem is a binary classification task aimed at predicting whether a customer will make a specific transaction in the future. Given historical data on customer transactions, Santander seeks to develop a predictive model to identify customers likely to engage in the specified transaction, regardless of the transaction amount.

Motivation:

Santander's motivation for addressing this problem stems from its mission to help people and businesses prosper by providing tailored financial solutions. By accurately predicting customer transactions, Santander can optimize its marketing strategies, offer targeted product recommendations, and enhance customer engagement. Additionally, by leveraging machine learning algorithms, Santander aims to improve the efficiency and effectiveness of its operations, ultimately leading to better customer outcomes and business performance.

Evaluation Metrics:

1. Weighted F1 score
2. Accuracy

3. Micro and Macro precision and recall

The contribution of each team member

Name	contribution
Peter Atef	Adaboost - Data Analysis - Grid Search Analysis - Base model
Bemoi Erian	XGboosting - Random Forest - Partial Dependencies Analysis
Mark Yasser	Logistic Regression - Feature Importance Analysis - hyper parameters Analysis
Karim Mahmoud	SVM - Learning Curve Analysis - Bias & Variance Analysis

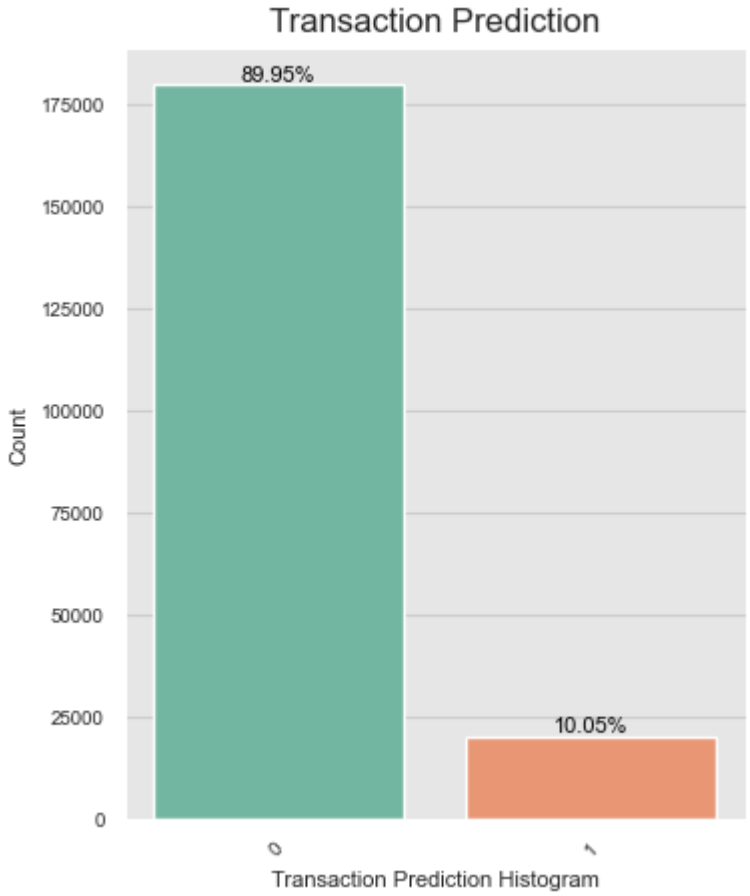
Exploratory Data Analysis (EDA)

Links to the dataset

Santander Customer Transaction Prediction

The dataset size is 100K and the number of features is 200

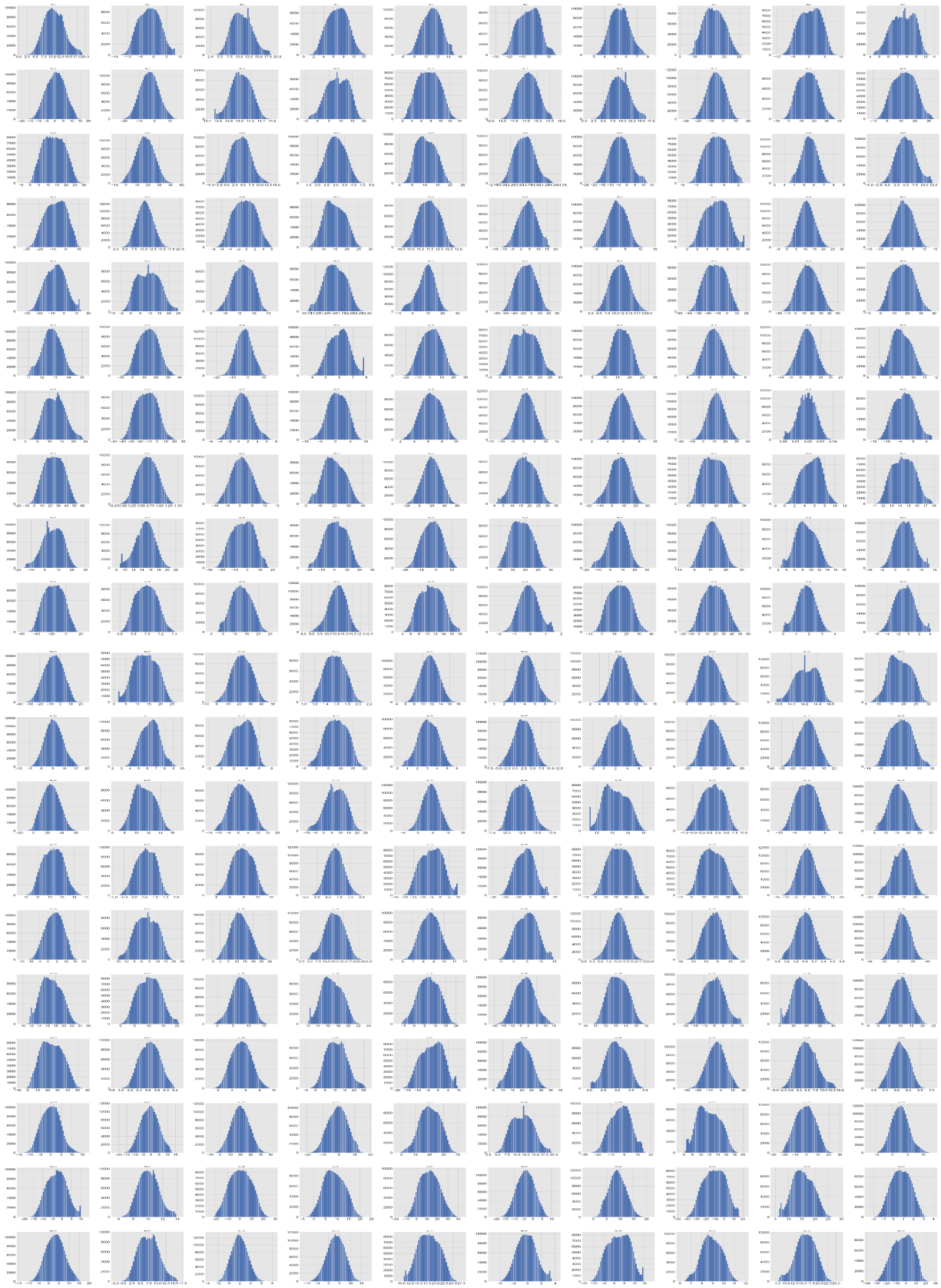
Target Variable Analysis



We can see that the number of points in class zero is way larger than the number of points in class one.

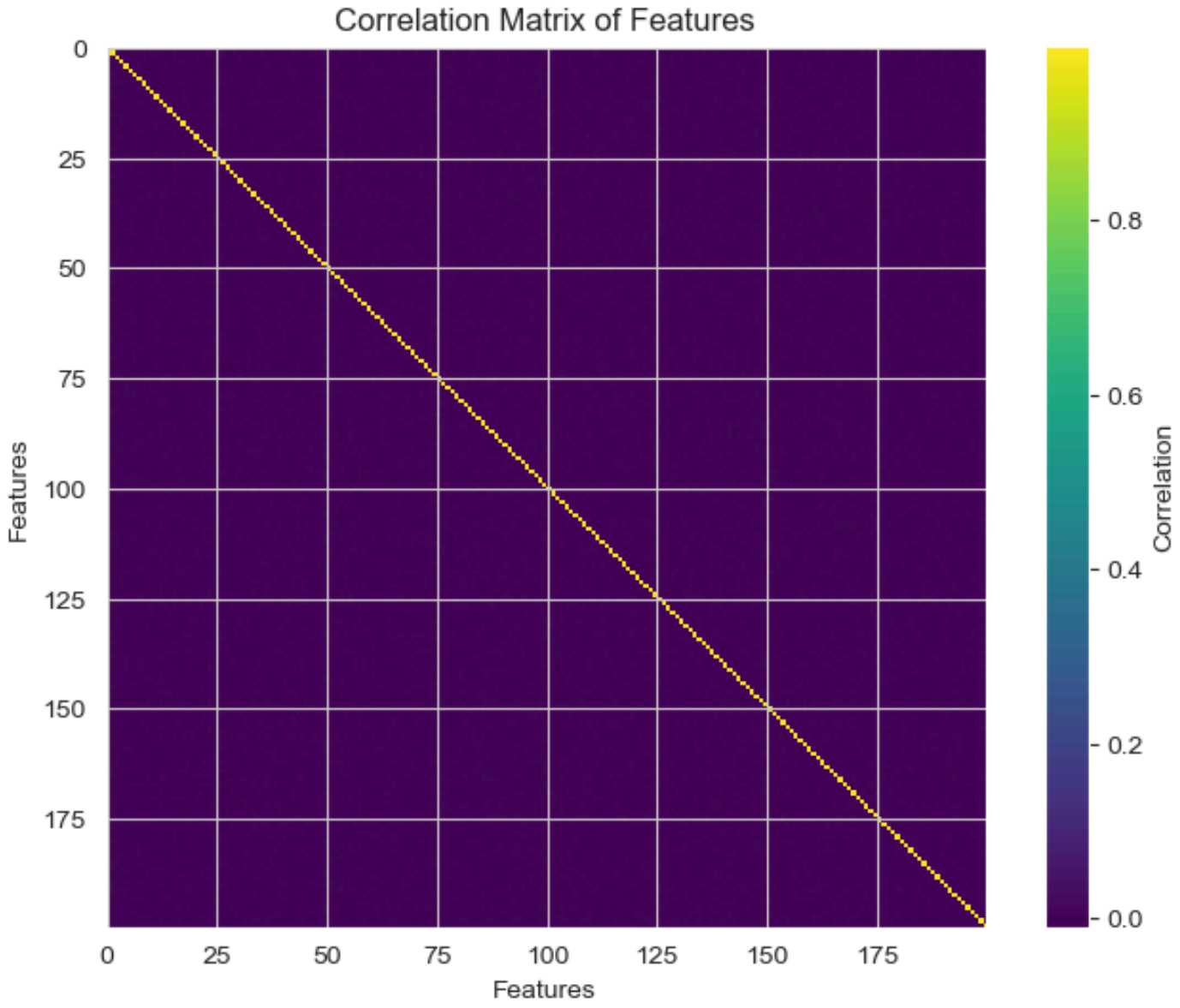
Features Distribution

The following graphs show the distribution of the features:



We can see that all features are normally distributed so using this information we can standardize the features by applying the following equation for each feature independently: X = (X - mean) / std
so that the mean of each feature will be almost zero and the variance almost one.

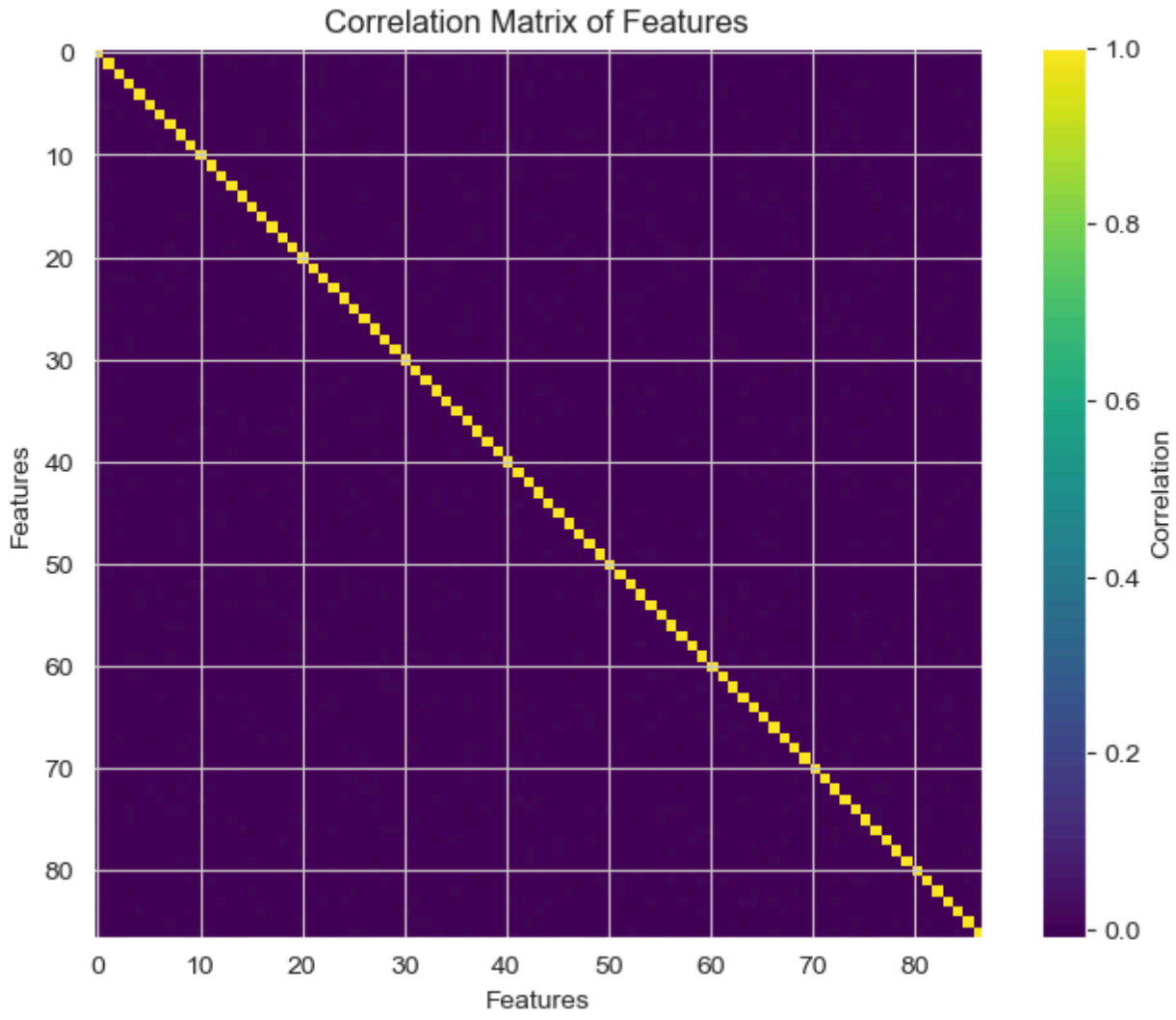
Features Correlation



We can see that all the features are almost independent of each other.

Variance Thresholding

After getting the variance of each feature the mean of variances will be 0.999, using this value we will threshold the features by their variance and the result is getting only 87 features out of 200. The following graph shows the correlation between the 87 features:

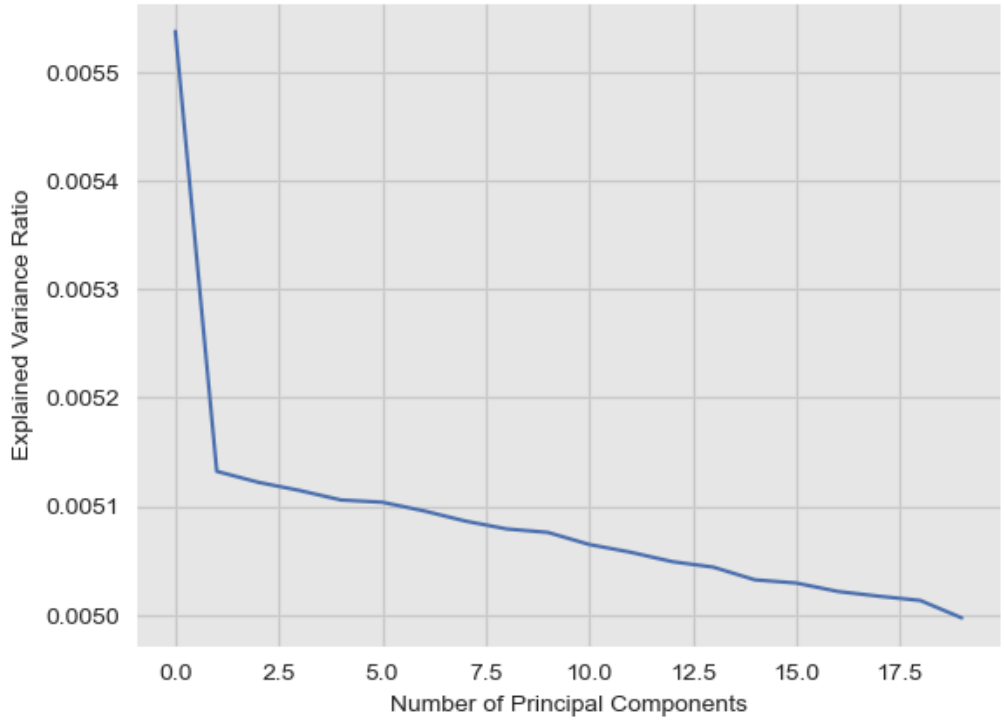


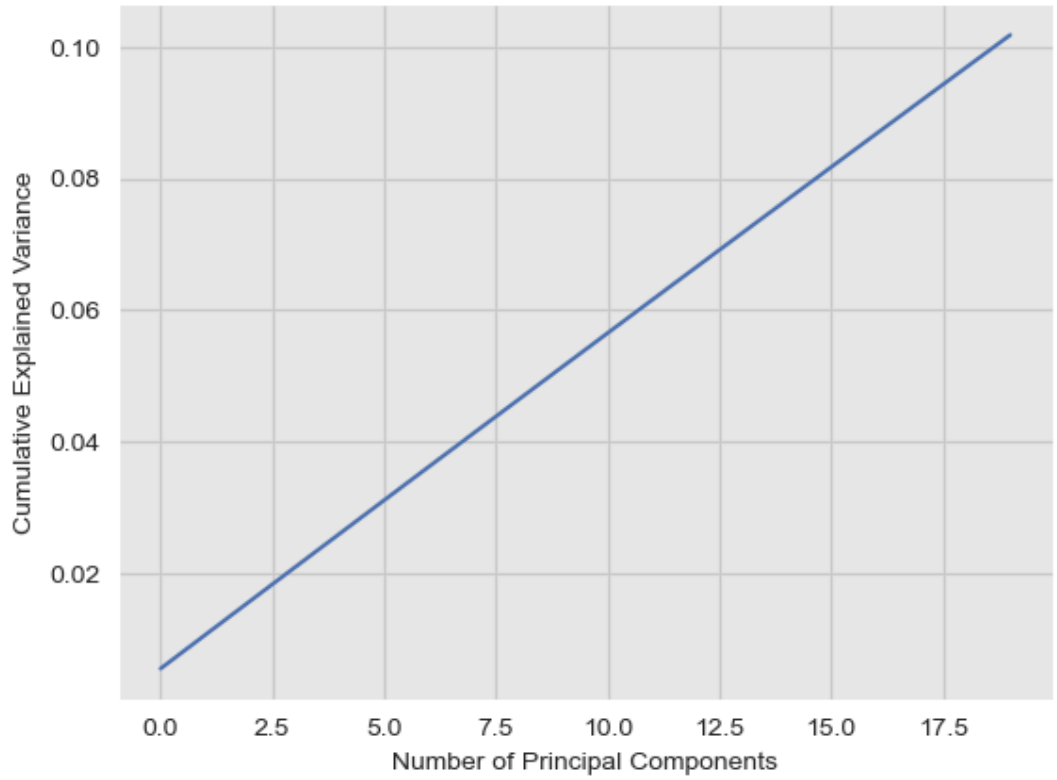
We can notice that the features are still independent of each other.

PCA

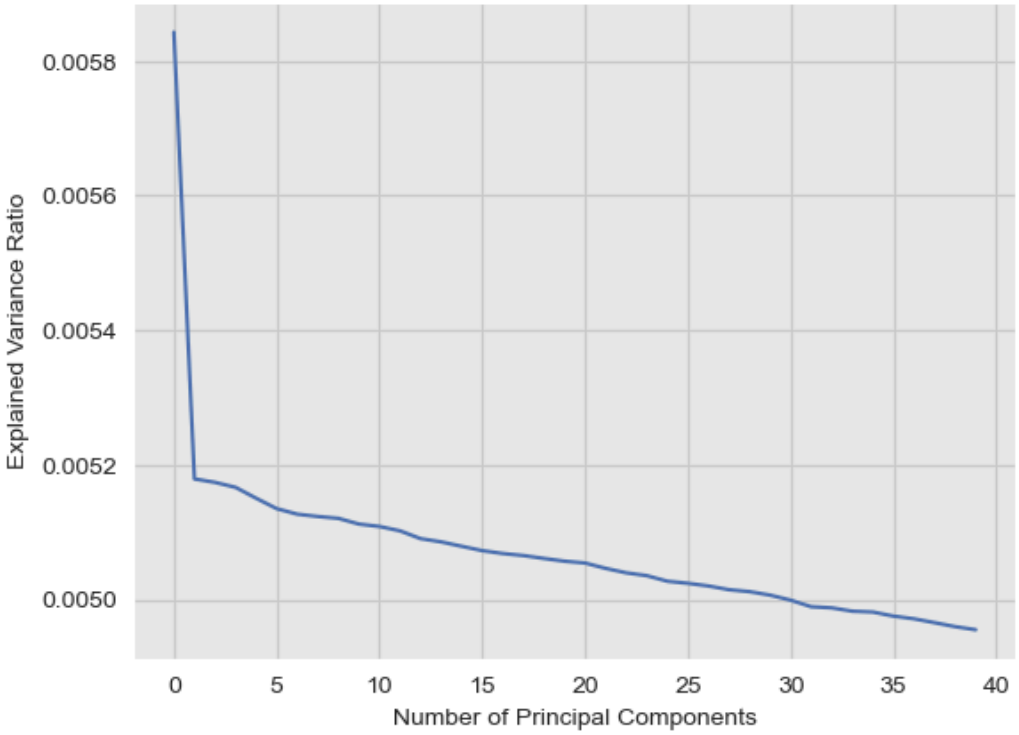
We applied PCA to get n components where 20<=n<=180 and we got the following results:

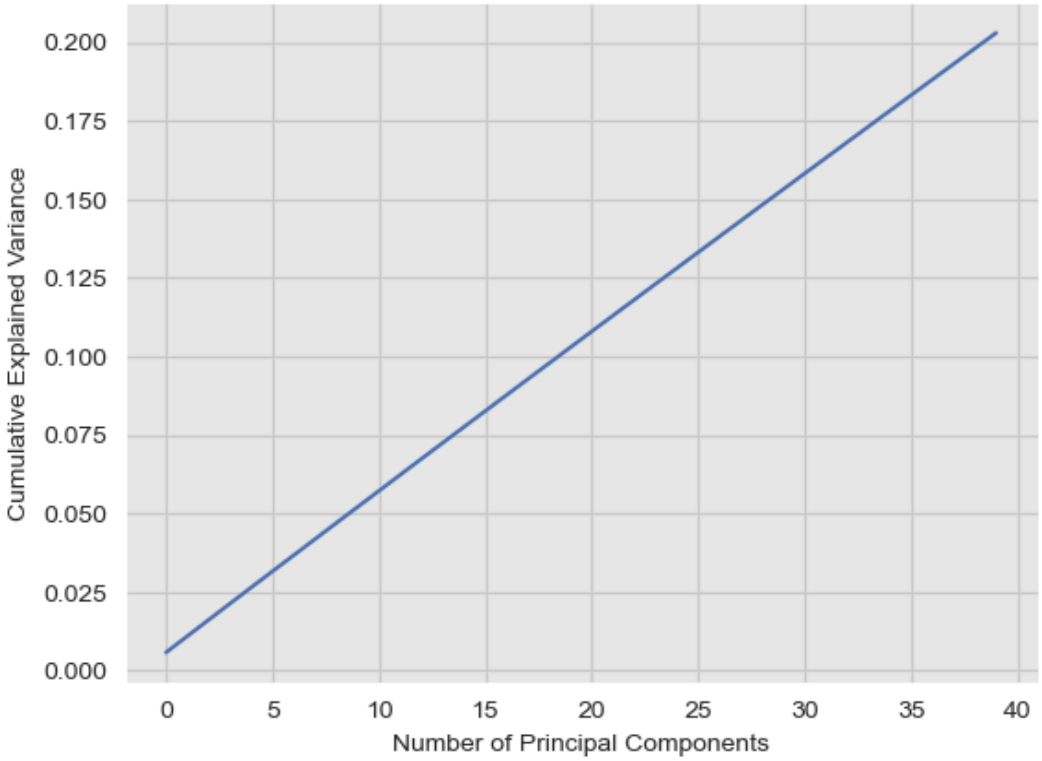
For n = 20


0.0055
0.0054
0.0053
0.0052
0.0051
0.0050
Explained Variance Ratio
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Number of Principal Components

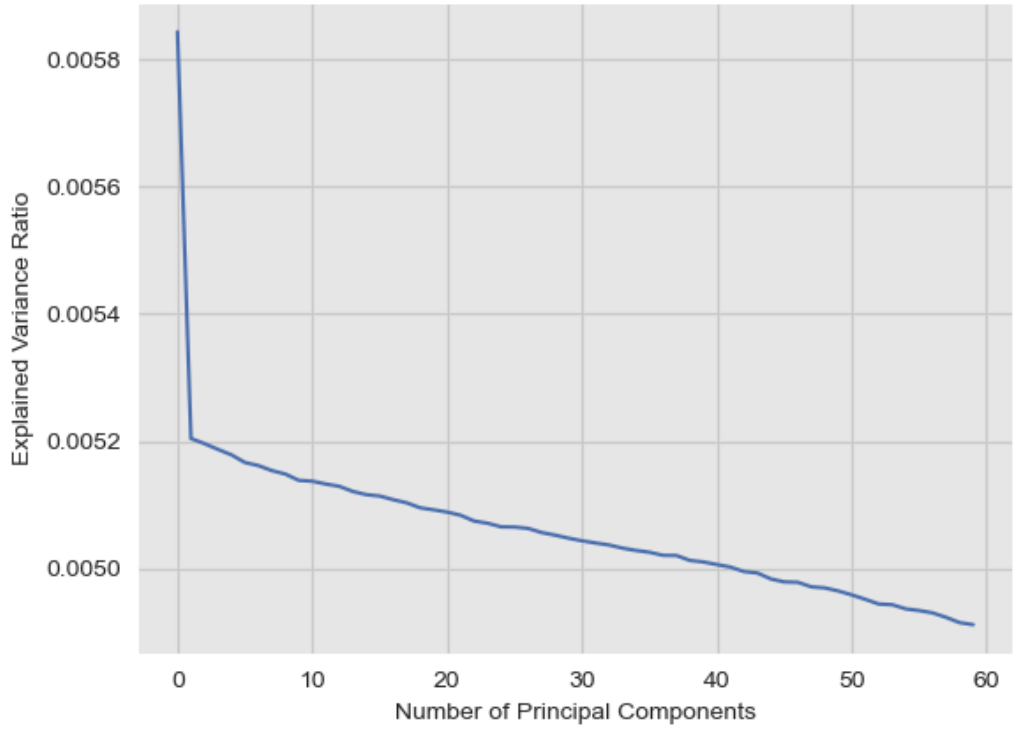

0.10
0.08
0.06
0.04
0.02
Cumulative Explained Variance
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Number of Principal Components

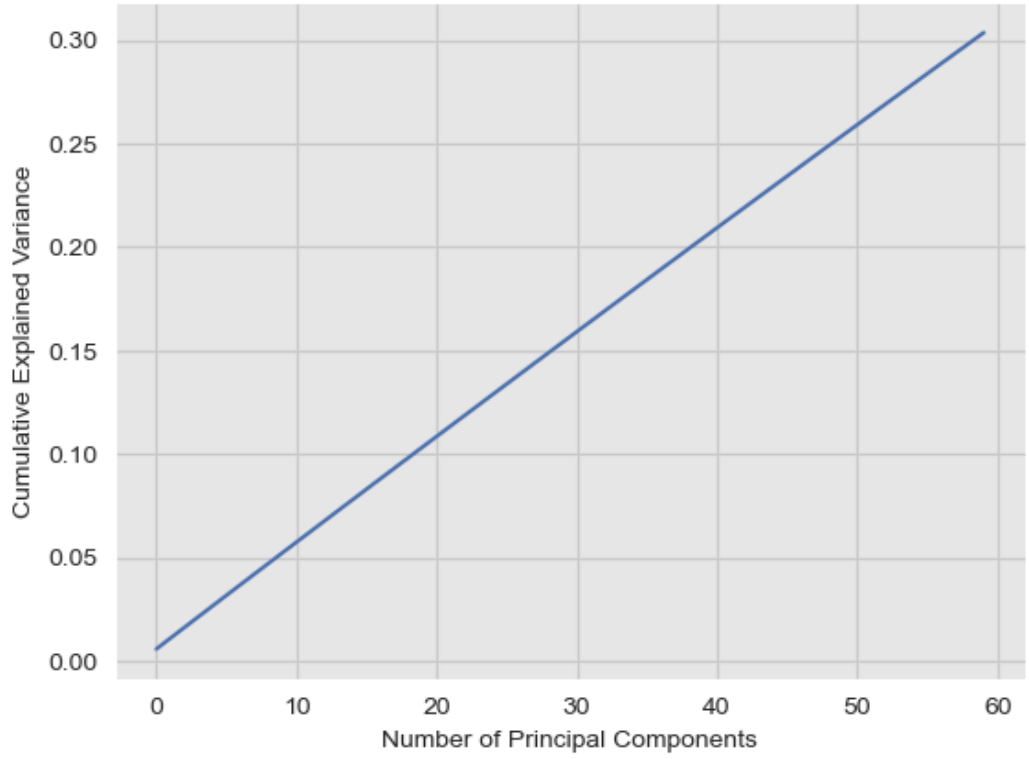
For n = 40


Explained Variance Ratio
0.0058
0.0056
0.0054
0.0052
0.0050
0
5
10
15
20
25
30
35
40
Number of Principal Components

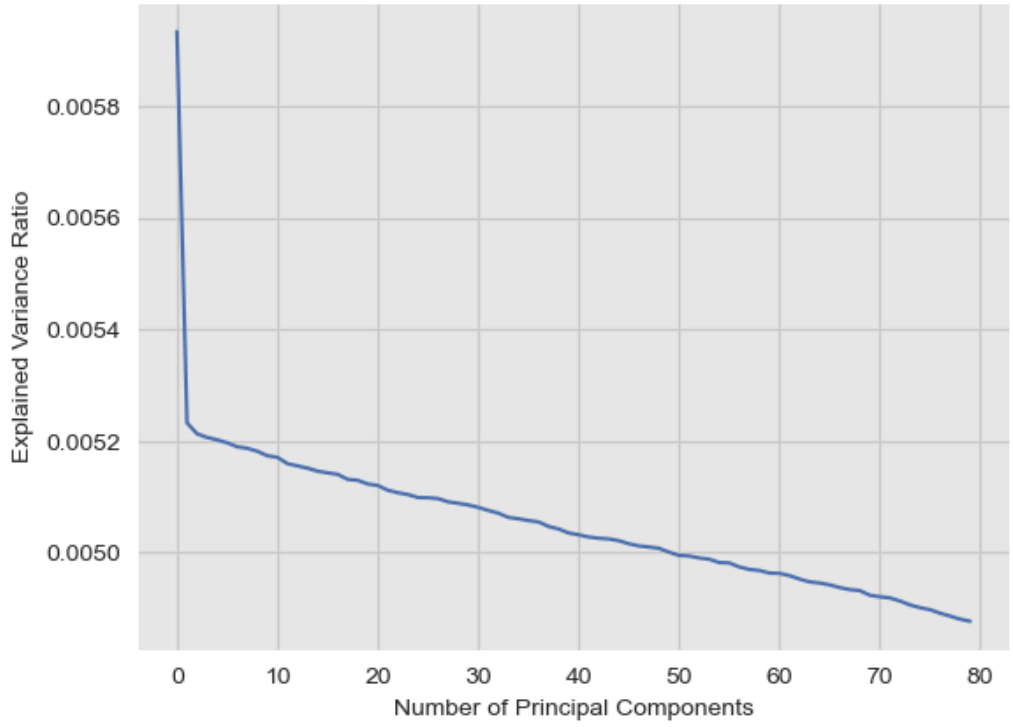

Cumulative Explained Variance
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
5
10
15
20
25
30
35
40
Number of Principal Components

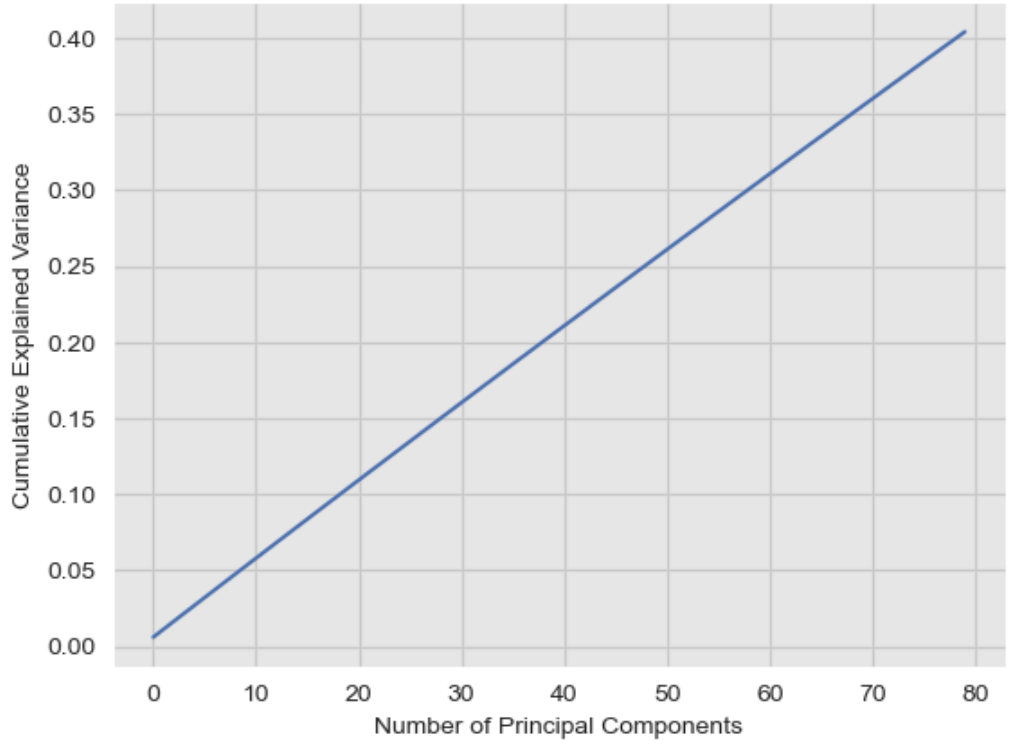
For n = 60


0.0058
0.0056
0.0054
0.0052
0.0050
Explained Variance Ratio
0
10
20
30
40
50
60
Number of Principal Components

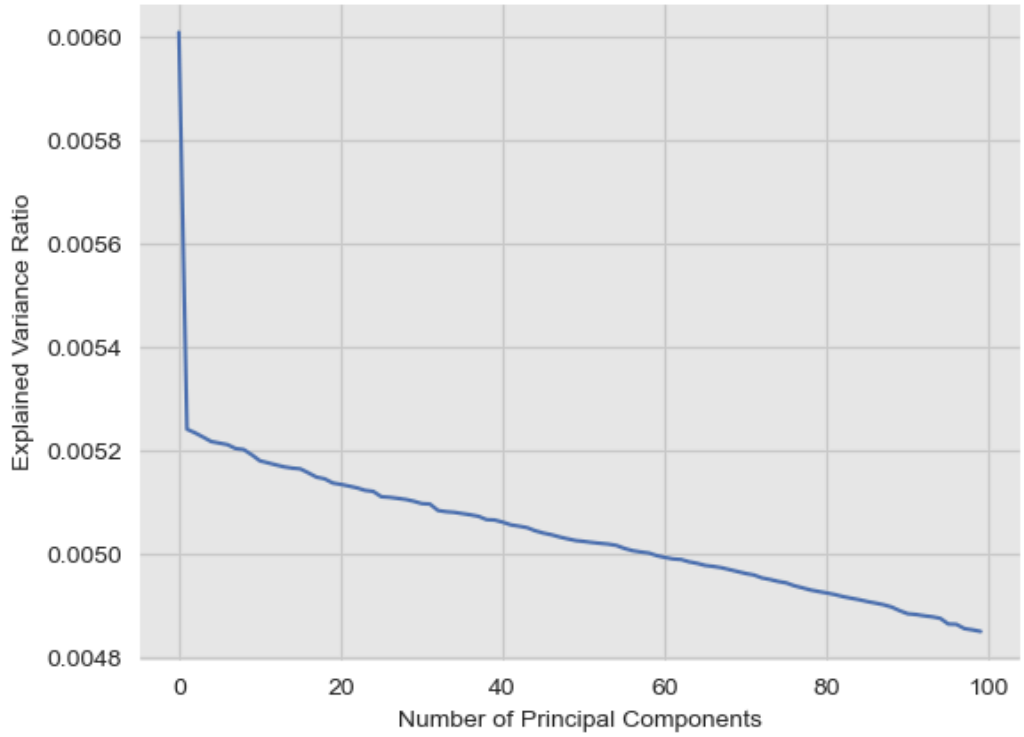

0.30
0.25
0.20
0.15
0.10
0.05
0.00
Cumulative Explained Variance
0
10
20
30
40
50
60
Number of Principal Components

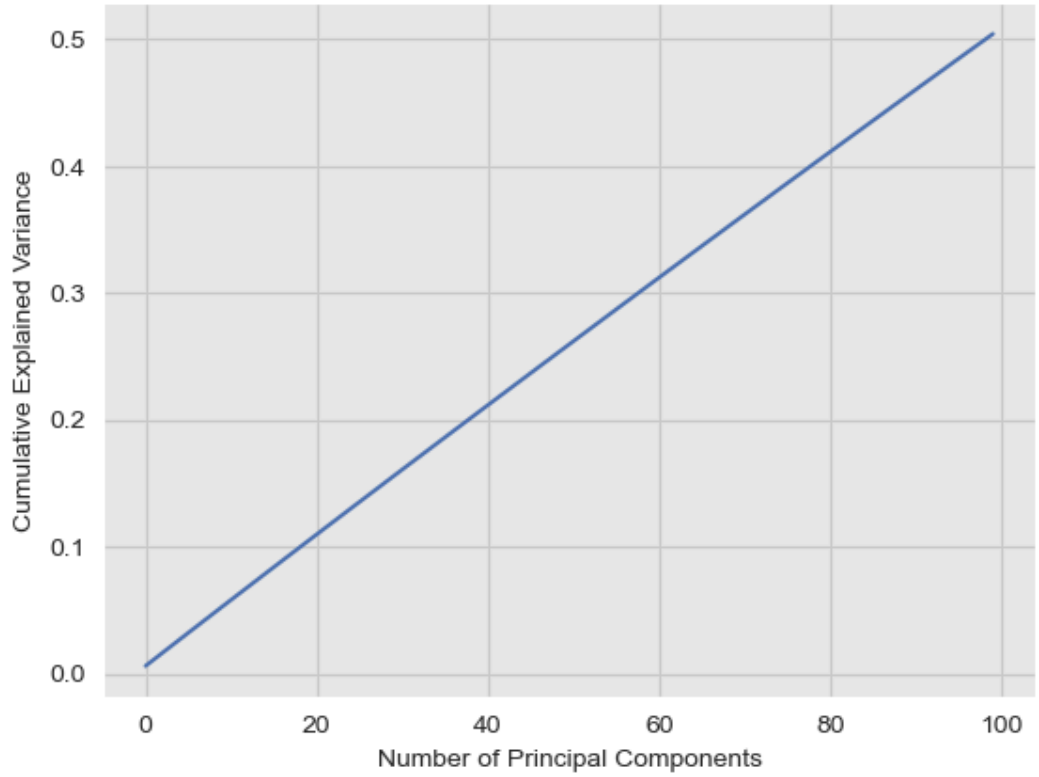
For n = 80


Explained Variance Ratio
0.0058
0.0056
0.0054
0.0052
0.0050
0
10
20
30
40
50
60
70
80
Number of Principal Components


Cumulative Explained Variance
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
10
20
30
40
50
60
70
80
Number of Principal Components

For n = 100


0.0060
0.0058
0.0056
0.0054
0.0052
0.0050
0.0048
Explained Variance Ratio
0
20
40
60
80
100
Number of Principal Components


0.5
0.4
0.3
0.2
0.1
0.0
Cumulative Explained Variance
0
20
40
60
80
100
Number of Principal Components

From the previous analysis, we can conclude that most of the variance ratio is concentrated in the first 20 features to maximize the number of points we can use for training we will use the minimum number of features because there will not be a huge loss in variance ratio.
Note that we needed to reduce the dimensionality of the space for 2 reasons:

1. To avoid the problem of the curse of dimensionality
2. It takes a lot of time for SVM and ensemble models to train and apply cross-validation algorithms and we want to maximize the number of points we use so we need to reduce the number of features to achieve that.

So our final dataset used for training is 100K rows and 20 features, each feature is normally distributed.

Models Analysis

All models are validated using cross-validation with 10 folds.

Base Model

Before working on developing sophisticated machine learning models, we used some dummy models as a baseline model to compare the performance of more sophisticated models. By comparing the performance of a complex model to that of a simple model, we can determine if the complex model is actually providing useful predictions or if it is overfitting the data. Dummy models also help identify if the problem has any inherent bias or if the dataset is imbalanced. Overall, starting with a dummy model is a good way to get a baseline understanding of the data and the problem before moving on to more complex models. We have four strategies:

1. **Stratified:**
    a. **Description:** The predict_proba method randomly samples one-hot vectors from a multinomial distribution parametrized by the empirical class prior probabilities. The predict method returns the class label which got probability one in the one-hot

vector of predict_proba. Each sampled row of both methods is therefore independent and identically distributed.

b. **Cross-validation:**
   i. accuracy: 0.8184699999999999
   ii. f1_macro: 0.4977632366599372
   iii. f1_micro: 0.8184699999999999

c. **Prediction results:**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.90      0.90     89917
           1       0.10      0.10      0.10     10083

    accuracy                           0.82    100000
   macro avg       0.50      0.50      0.50    100000
weighted avg       0.82      0.82      0.82    100000
```

2. **most_frequent:**
   a. **Description:** The predict method always returns the most frequent class label in the observed y argument passed to fit. The predict_proba method returns the matching one-hot encoded vector.
   b. **Cross-validation:**
      i. accuracy: 0.8998500000000001
      ii. f1_macro: 0.47364265563458696
      iii. f1_micro: 0.8998500000000001
   c. **Prediction results:**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      1.00      0.95     89917
           1       0.00      0.00      0.00     10083

    accuracy                           0.90    100000
   macro avg       0.45      0.50      0.47    100000
weighted avg       0.81      0.90      0.85    100000
```

3. **uniform**
    a. **Description:** Generates predictions uniformly at random from the list of unique classes observed in y, i.e. each class has equal probability.
    b. **Cross-validation:**
        i. accuracy: 0.5008999999999999
        ii. f1_macro: 0.4056152067337889
        iii. f1_micro: 0.5008999999999999
    c. **Prediction results:**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.50      0.64     89917
           1       0.10      0.49      0.17     10083

    accuracy                           0.50    100000
   macro avg       0.50      0.50      0.40    100000
weighted avg       0.82      0.50      0.59    100000
```

4. **constant;**
    a. **Description:** Always predicts a constant label that is provided by the user. This is useful for metrics that evaluate a non-majority class.
    b. **Cross-validation:**
        i. accuracy: 0.8998500000000001
        ii. f1_macro: 0.47364265563458696
        iii. f1_micro: 0.8998500000000001

c. **Prediction results:**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      1.00      0.95     89917
           1       0.00      0.00      0.00     10083

    accuracy                           0.90    100000
   macro avg       0.45      0.50      0.47    100000
weighted avg       0.81      0.90      0.85    100000
```

5. **prior:**
   a. **Description:** The predict method always returns the most frequent class label in the observed y argument passed to fit (like most_frequent) but Predict_proba always returns the empirical class distribution of y also known as the empirical class prior distribution.
   b. **Cross-validation:**
      i. accuracy: 0.8998500000000001
      ii. f1_macro: 0.47364265563458696
      iii. f1_micro: 0.8998500000000001
   c. **Prediction results:**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      1.00      0.95     89917
           1       0.00      0.00      0.00     10083

    accuracy                           0.90    100000
   macro avg       0.45      0.50      0.47    100000
weighted avg       0.81      0.90      0.85    100000
```

Logistic regression

Logistic regression is a powerful tool for predicting categorical outcomes. It is used in a wide variety of fields, including marketing, medicine, and

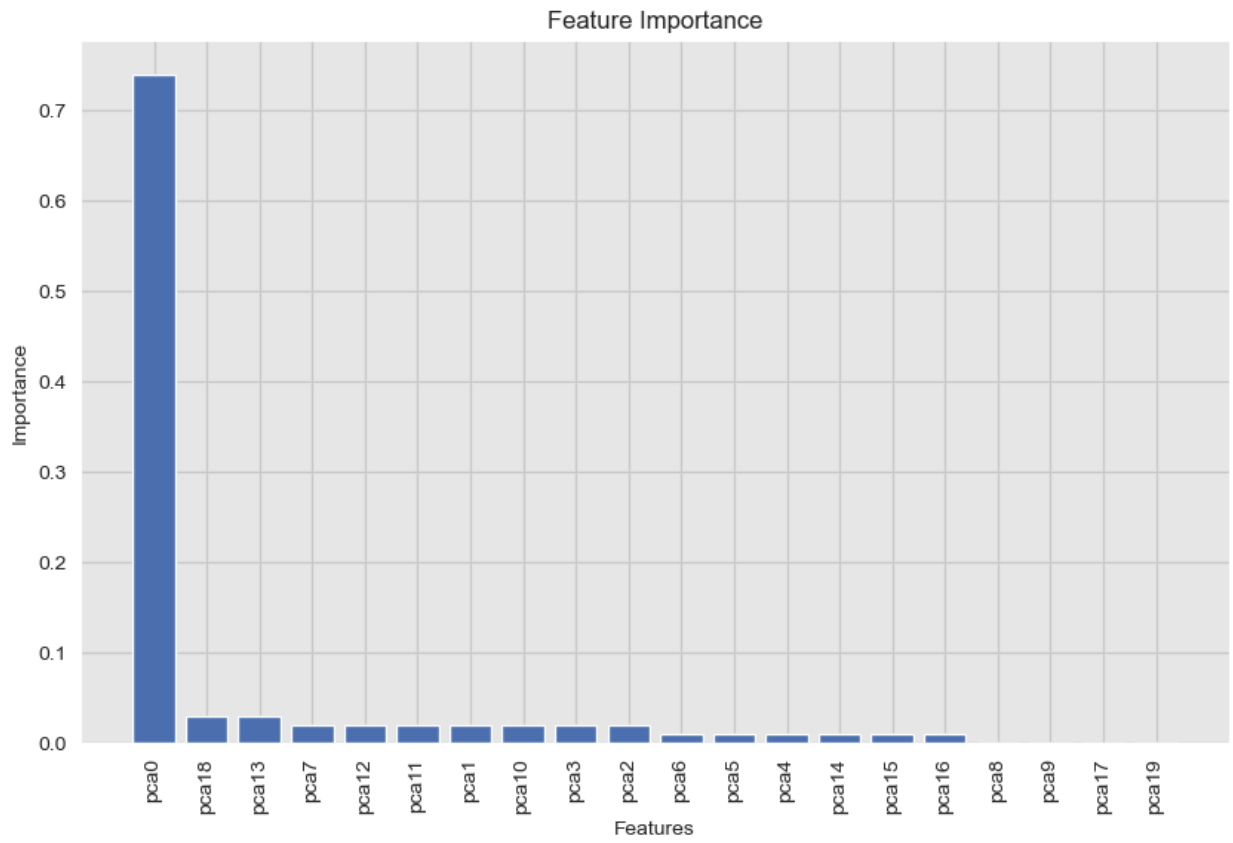
finance. For example, logistic regression can be used to predict the likelihood that a customer will buy a product, the likelihood that a patient will develop a disease or the likelihood that a company will go bankrupt.

- Advantages:
    - It is a simple method to predict categorical outcomes.
    - It can be used to predict the probability of an outcome for any given combination of predictor values.
    - It is relatively easy to interpret the results of a logistic regression model.
- Disadvantages:
    - It can be sensitive to outliers in the data.
    - It can be difficult to interpret the results of a logistic regression model when there are multiple independent variables.
    - It can be computationally expensive to fit a logistic regression model with a large number of independent variables.

Overall, logistic regression is a powerful tool for predicting categorical outcomes. It is relatively easy to use and interpret, and it can be used in a wide variety of fields.

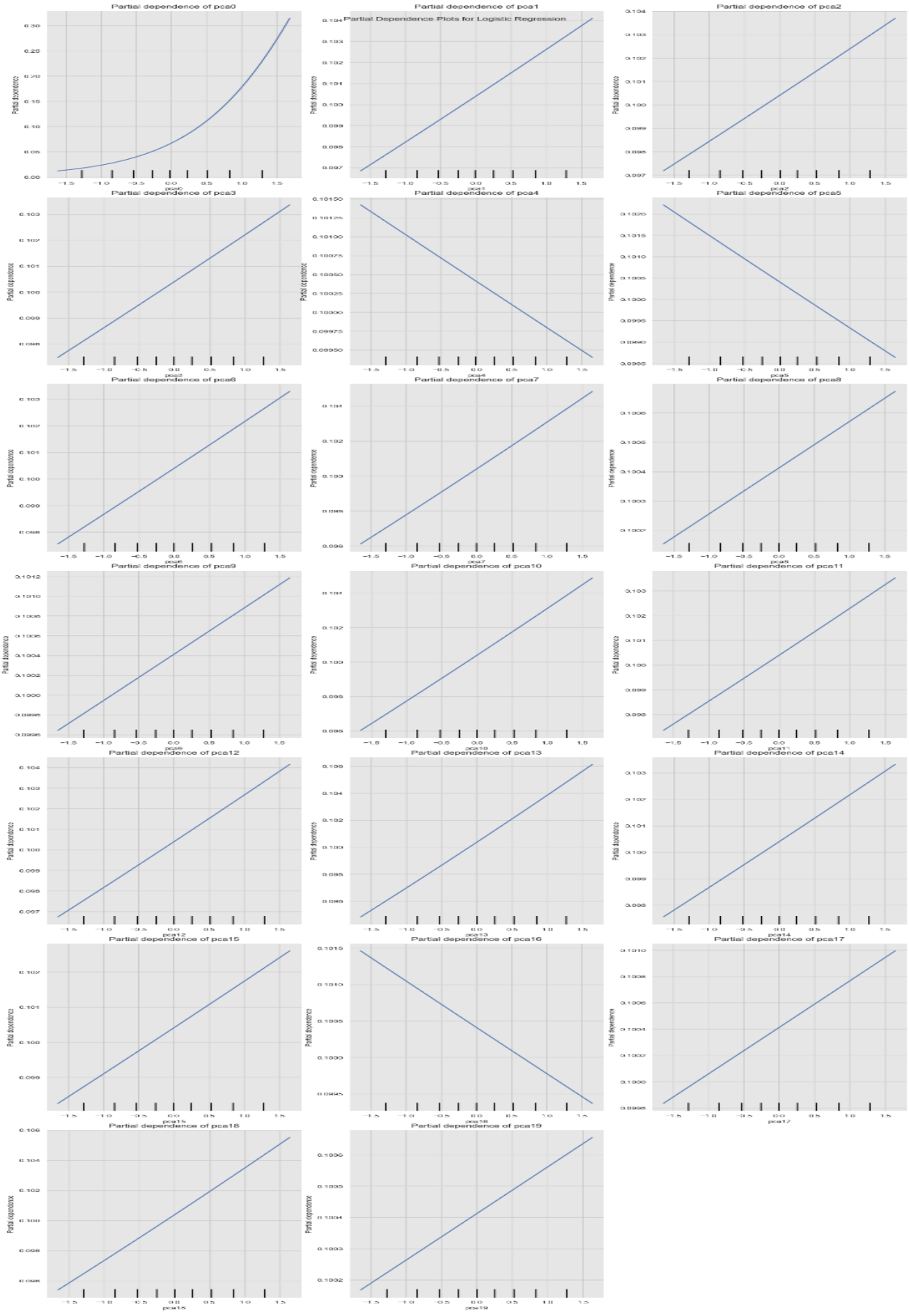
Feature Importance Plot

Note that we now only use 20 features that we got from PCA to be able to train and apply cross-validation on all models.



Based on the previous plot we can see that the first feature is the most important one with 75% importance and the other 19 features have an importance of 25%. So we may reduce the number of features to ten instead of twenty if you will use logistic regression to solve this problem.

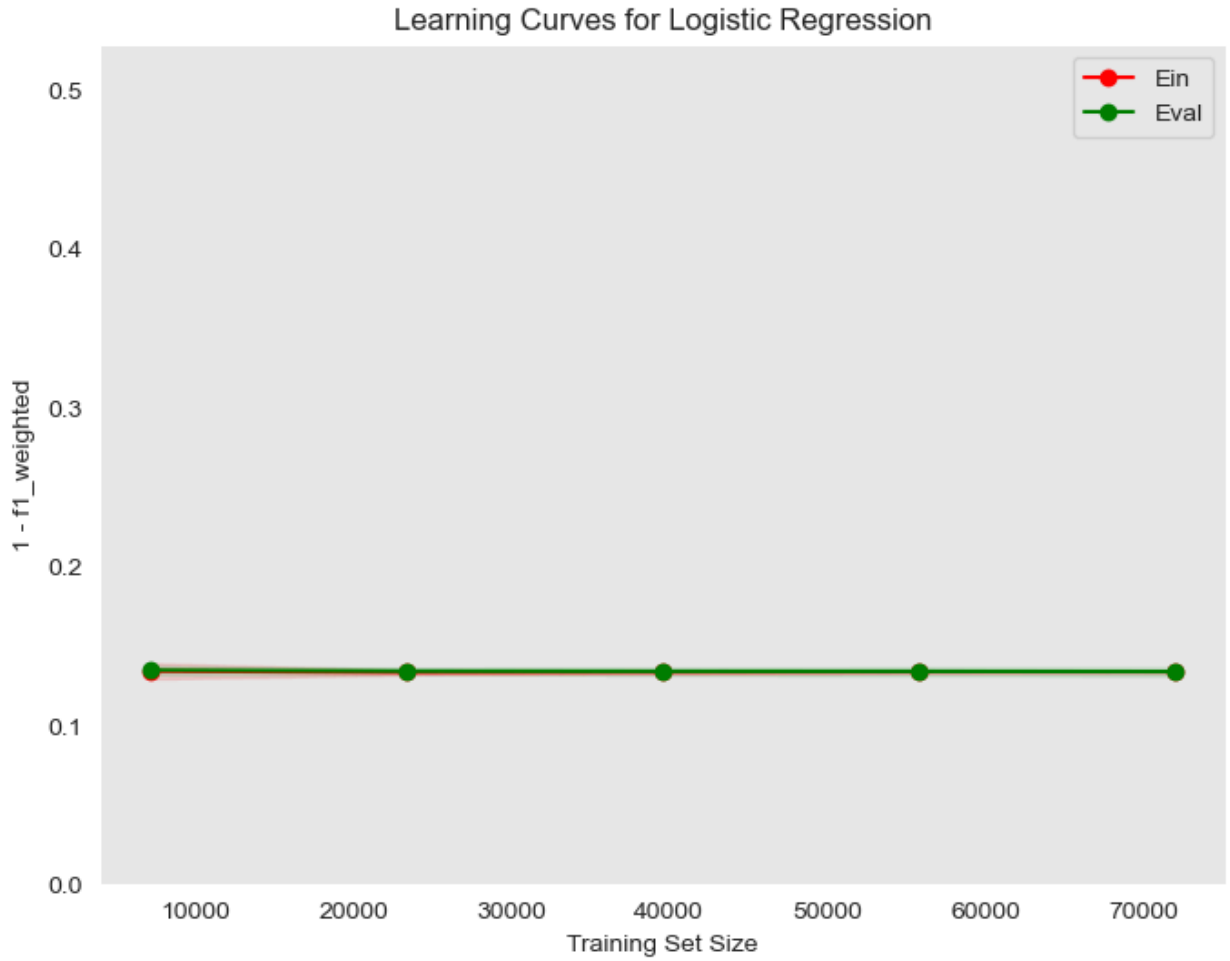
Partial Dependence Analysis



We can see that for the first feature the relationship isn't linear but for the rest of the features, it's a linear relationship either direct or inversely proportional.

Learning Curves Plot

show the training error (Ein) and validation error (Eval) as a function of the training set size



We can see that the training error is near to the validation error so the model can generalize pretty well.

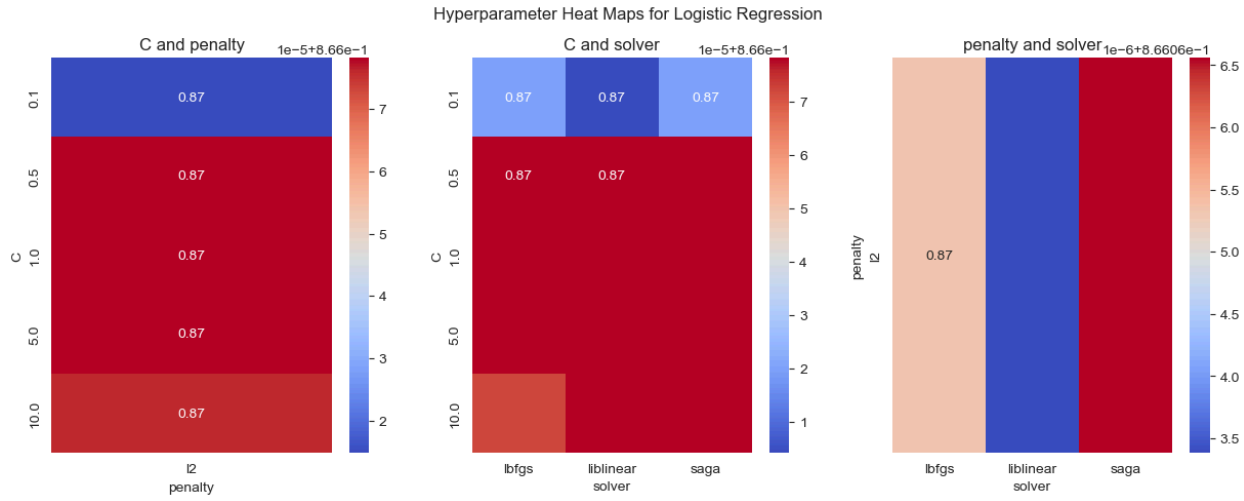
Hyperparameter Tuning

This is a process of adjusting the parameters of a model to optimize its performance. It can be done using techniques like grid search, random search, or Bayesian optimization.

Grid Search

```
param_grid = {
    'C': [0.1, 0.5, 1, 5, 10],
    'penalty': ['l2'],
    'solver': ['lbfgs', 'liblinear', 'saga']
}
```

Where “C” is the inverse of regularization strength; smaller values specify stronger regularization, and “penalty” specifies the norm used in the penalization. It could be 'l1' for L1 regularization, 'l2' for L2 regularization, or 'none' for no regularization, and “solver” is the algorithm used in the optimization problem.
Here are some heatmap visualizations of the grid search results:



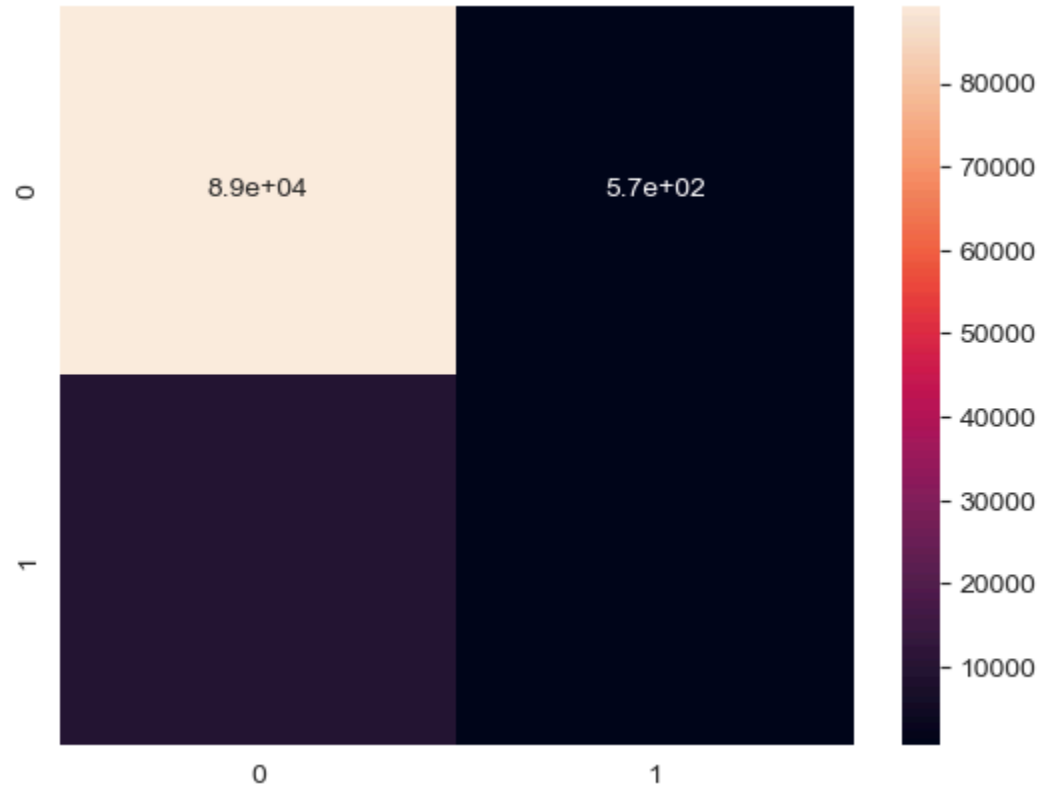
We can see that the “lbfgs” solver gives the highest accuracy with the “L2” penalty. Also, we can see that there are multiple combinations of “C” and “Solver” that work well together.

After applying the grid search we found that the best parameters are:

{'C': 0.5, 'penalty': 'l2', 'solver': 'lbfgs'}

After training a model using the best parameters we got from the search grid we got the following results:

Confusion matrix:

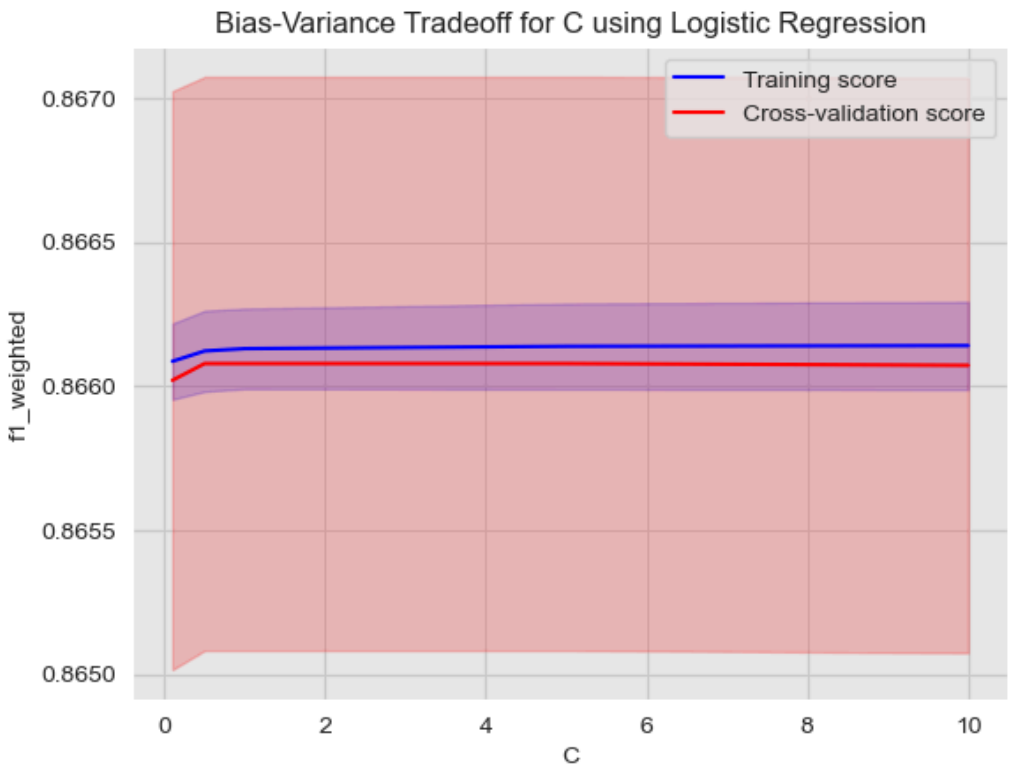


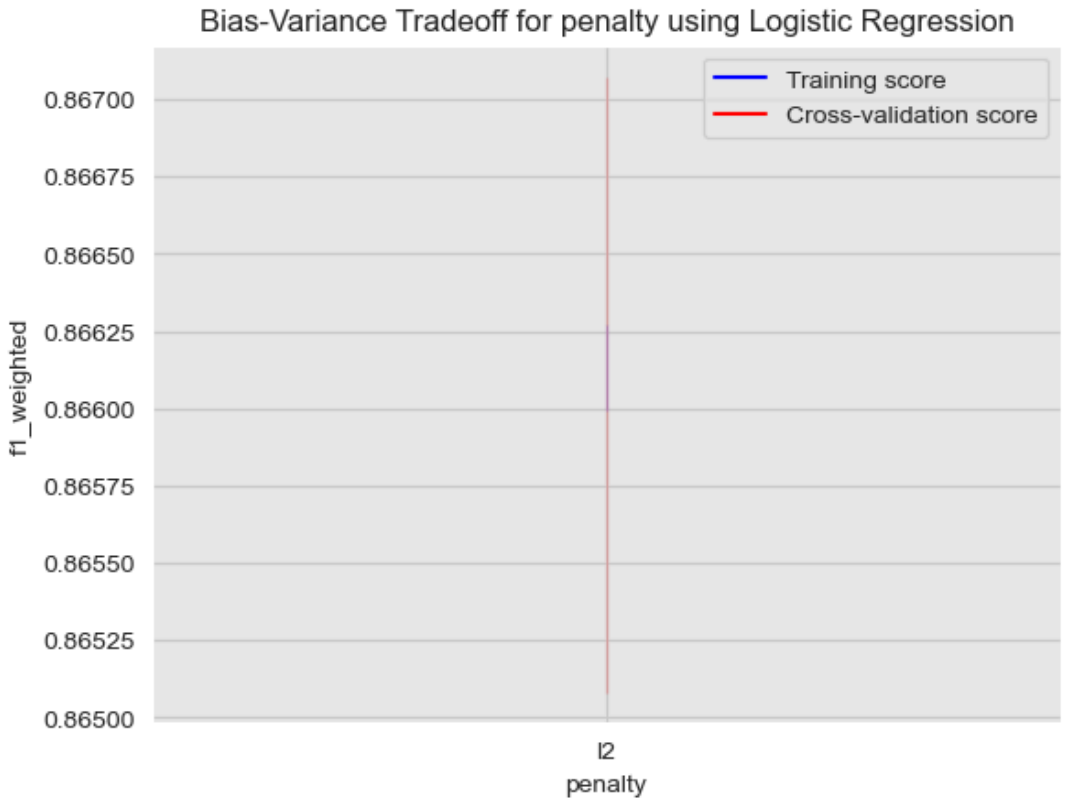
Classification report:

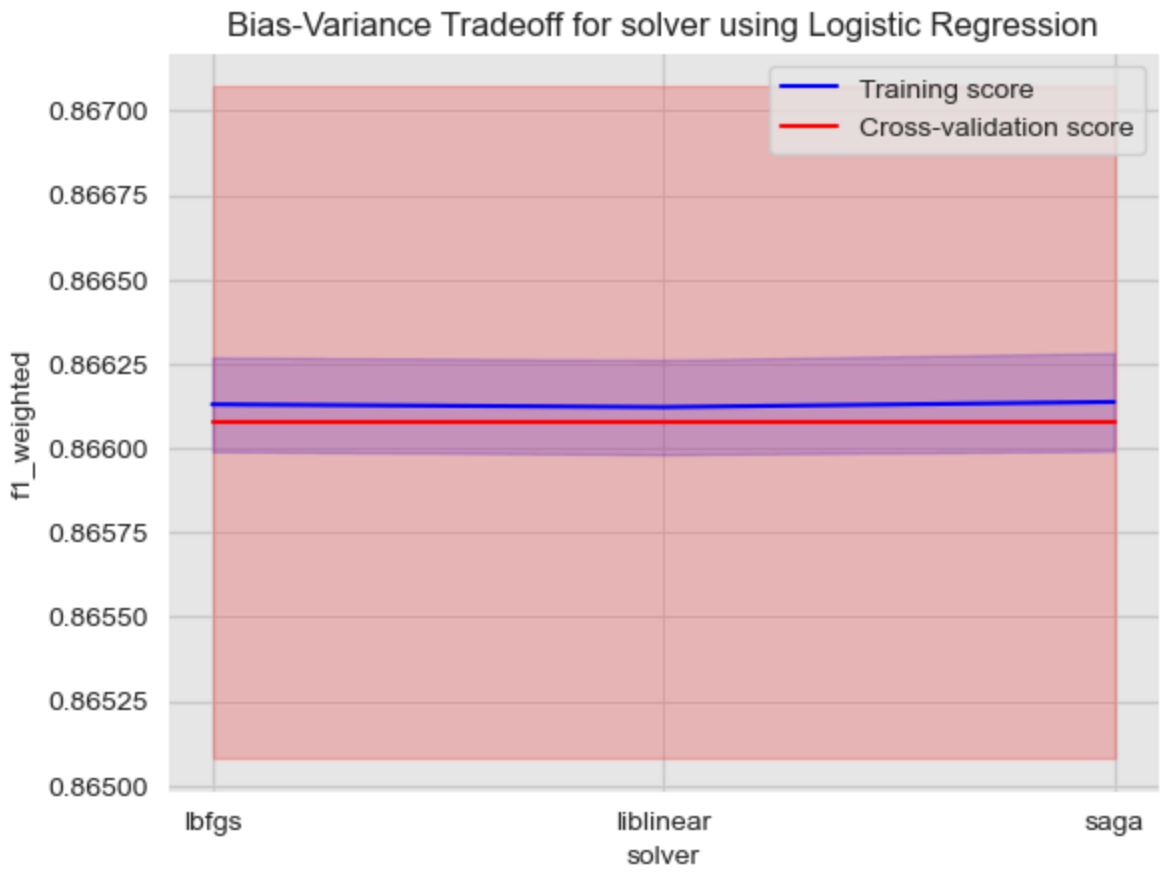
```
Classification Report of LogisticRegression(C=0.5, max_iter=10000, random_state=42) is:
              precision    recall  f1-score   support

           0       0.91      0.99      0.95     89917
           1       0.62      0.09      0.16     10083

    accuracy                           0.90    100000
   macro avg       0.76      0.54      0.55    100000
weighted avg       0.88      0.90      0.87    100000
```

Train-Validation Curve

Here are some Train-Validation Curves that we further used for the hyperparameter tuning process:


Bias-Variance Tradeoff for C using Logistic Regression
Training score
Cross-validation score
0.8670
0.8665
0.8660
0.8655
0.8650
f1_weighted
0
2
4
6
8
10
C


Bias-Variance Tradeoff for penalty using Logistic Regression
Training score
Cross-validation score
0.86700
0.86675
0.86650
0.86625
0.86600
0.86575
0.86550
0.86525
0.86500
f1_weighted
l2
penalty



Bias-variance Analysis

- mean square error: 0.0973067
- bias: 0.096430338
- var: 0.0008763620000000017
- Estimated Eout: 0.09730670000000001

In our problem, we can see that the model has a low bias and a low variance, which means that it is well-fitted to the data and can generalize well to new, unseen data because of the small Eout.

Support Vector Machines

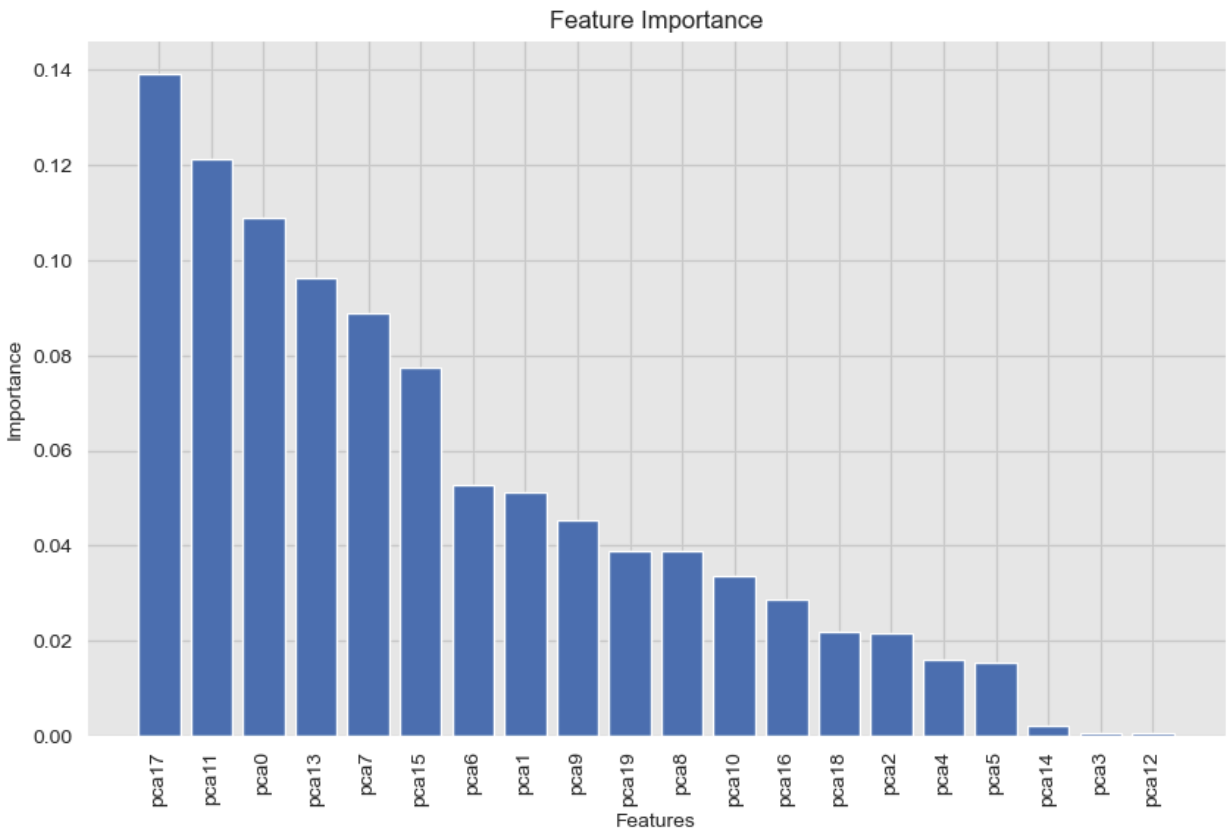
Support Vector Machines (SVMs) are a class of supervised learning algorithms used for classification and regression analysis. SVMs work by finding the hyperplane that best separates the data into different classes. The optimal decision boundary is the hyperplane that maximizes the margin between the two classes. In the case where the data is not linearly

separable, SVMs use a kernel trick to map the data into a higher-dimensional space where the data can be linearly separated.

- **Advantages of SVM:**
    - Effective in high-dimensional spaces: SVMs can perform well in high-dimensional spaces, making them useful for solving complex problems with many features.
    - Robust against overfitting: SVMs are less prone to overfitting than other classification algorithms because they try to maximize the margin between classes, which helps prevent the model from being too closely fit to the training data.
    - Versatile: SVMs can be used for both linear and nonlinear classification and regression tasks, and different kernel functions can be used to handle different types of data.
    - Works well with small and large datasets: SVMs are computationally efficient and can work well with small and large datasets.
- **Disadvantages of SVM:**
    - Can be sensitive to the choice of kernel: SVM performance depends heavily on the choice of kernel, which can be challenging to choose correctly.
    - Requires careful preprocessing of data: SVMs are sensitive to the scale of the input features, so data preprocessing is required to standardize the features.
    - Computationally intensive: SVMs can be computationally intensive, particularly for large datasets and complex kernel functions.
    - Binary classification only: SVMs are designed for binary classification problems, which means they need to be modified for multi-class classification tasks.

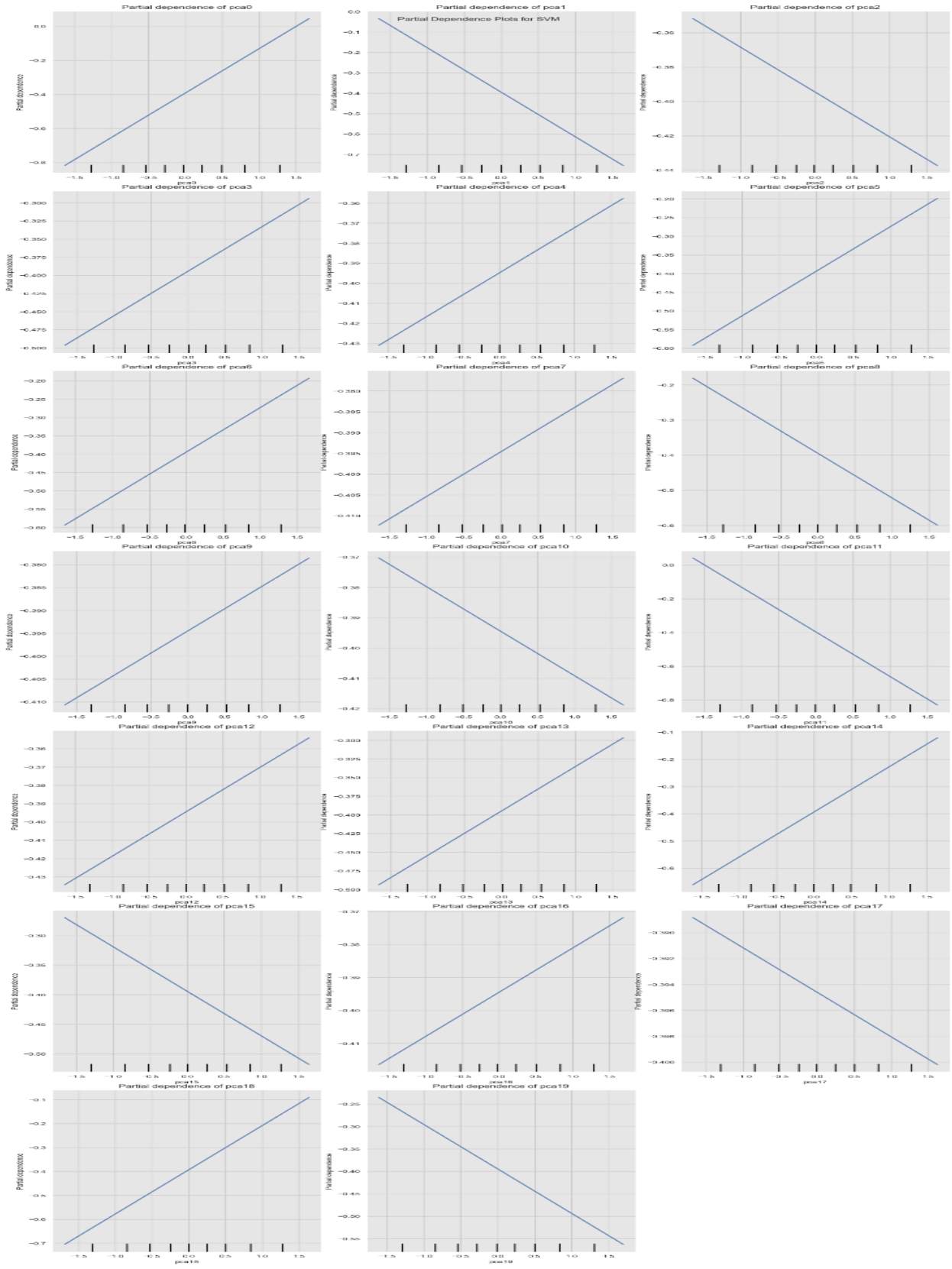
Feature Importance Plot

Note that we now only using 20 features that we got from PCA to be able to train and apply cross-validation on all models.



Based on the previous plot we can see that the 18th feature is the highest importance and the more we add features, the less importance they get. Unlike logistic regression, SVM makes use of most of the features.

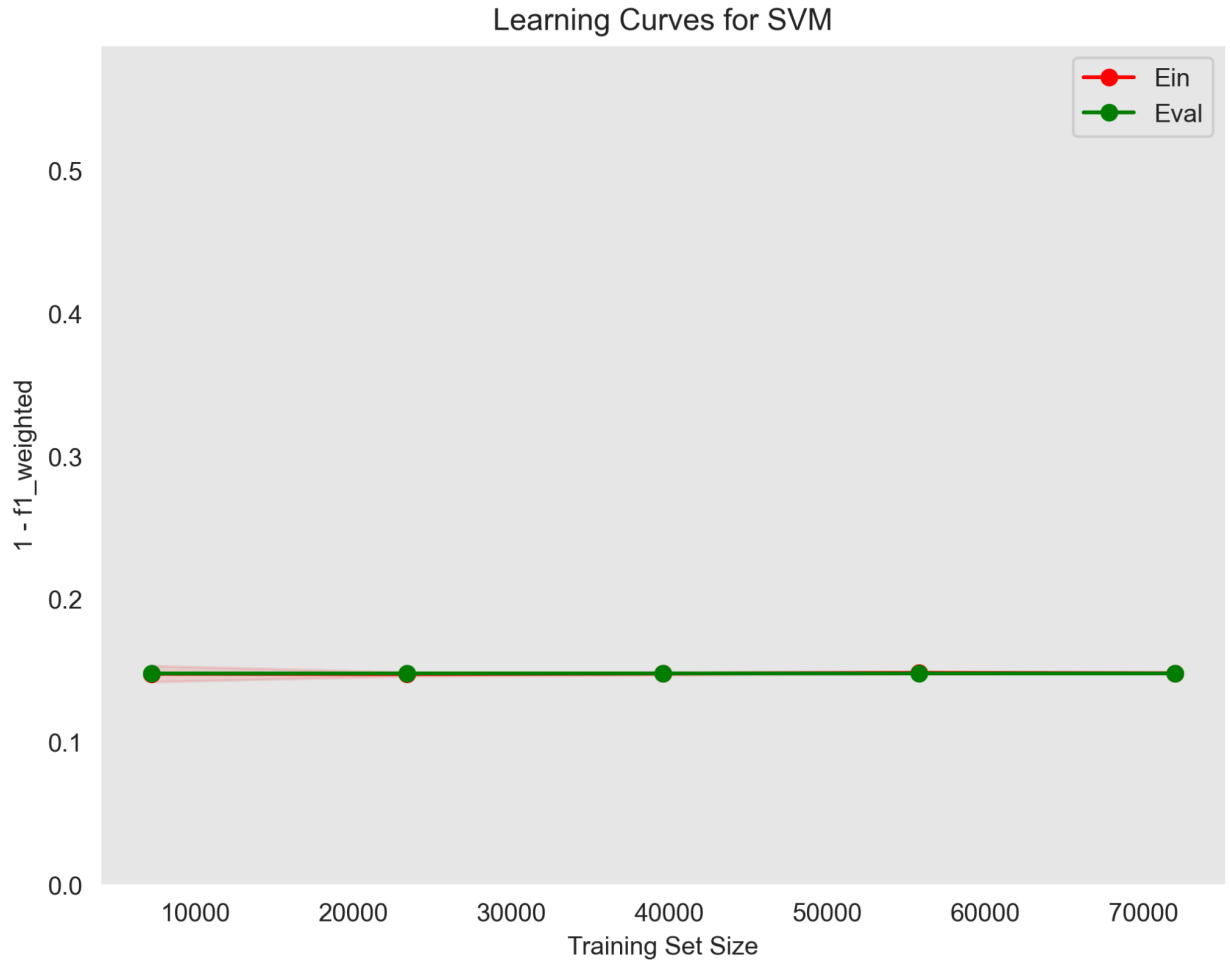
Partial Dependence Analysis



We can see that for the first feature the relationship isn't linear but for the rest of the features, it's a linear relationship either direct or inversely proportional.

Learning Curves Plot

shows the training error (Ein) and validation error (Eval) as a function of the training set size



We can see that the training error is near to the validation error so the model can generalize pretty well.

Hyperparameter Tuning

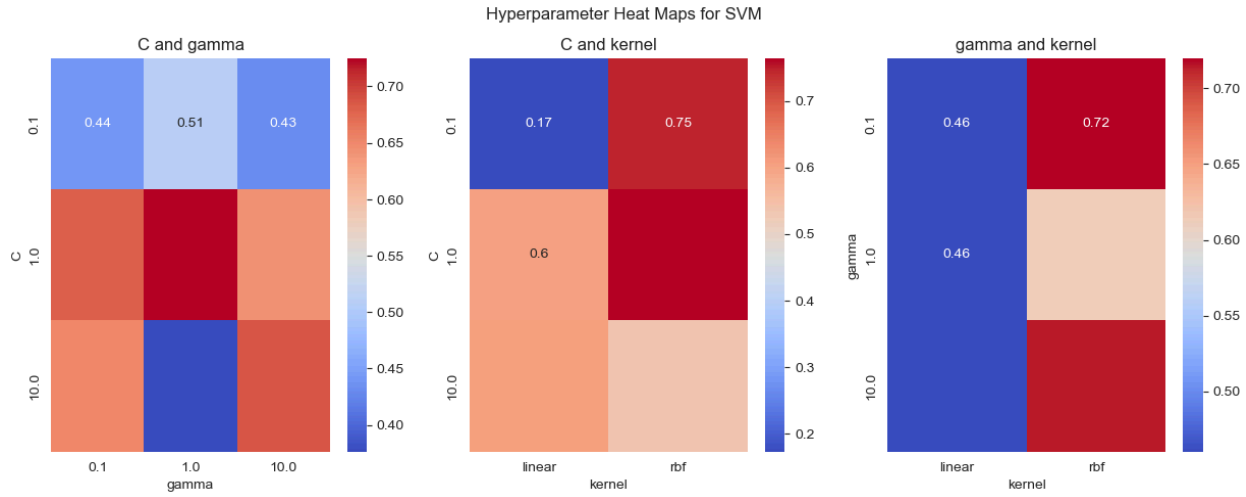
This is a process of adjusting the parameters of a model to optimize its performance. It can be done using techniques like grid search, random search, or Bayesian optimization.

Grid Search

```
# Define the parameter grid to search over
param_grid = {
    'C': [0.1, 1, 10],
    'gamma': [0.1, 1, 10],
    'kernel': ['rbf', 'linear']
}
```

Where “C” is the regularization parameter. It controls the trade-off between allowing the model to fit the training data well and keeping the decision boundary smooth. Larger values of C lead to less regularization, allowing the model to fit the training data more closely, “gamma” kernel coefficient for 'rbf', 'poly', and 'sigmoid'. Higher values of gamma lead to more complex decision boundaries., and “kernel” specifies the type of kernel used in the algorithm. It could be 'linear', 'poly', 'rbf' (radial basis function), 'sigmoid', or a custom kernel.

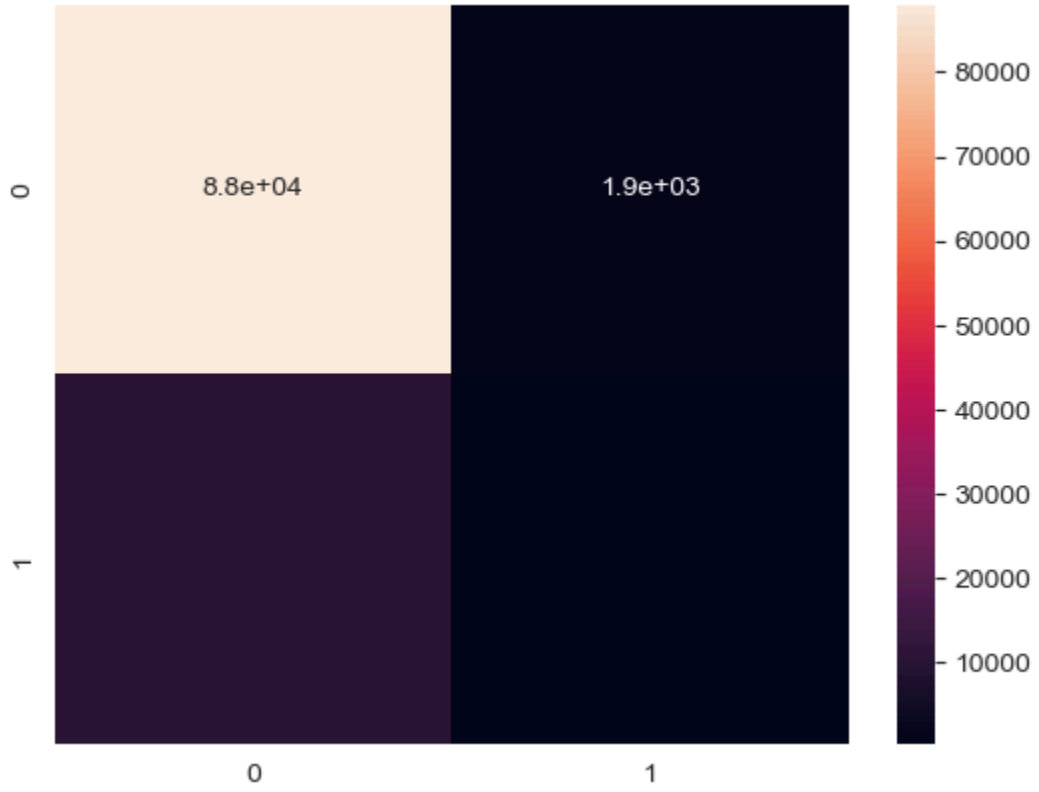
Here are some heatmap visualizations of the grid search results:



After applying the grid search we found that the best parameters are:

{'max_iter': 1000, 'random_state': 0, 'C': 1, 'gamma': 1, 'kernel': 'rbf'}

After training a model using the best parameters we got from the search grid we got the following results:
Confusion matrix:

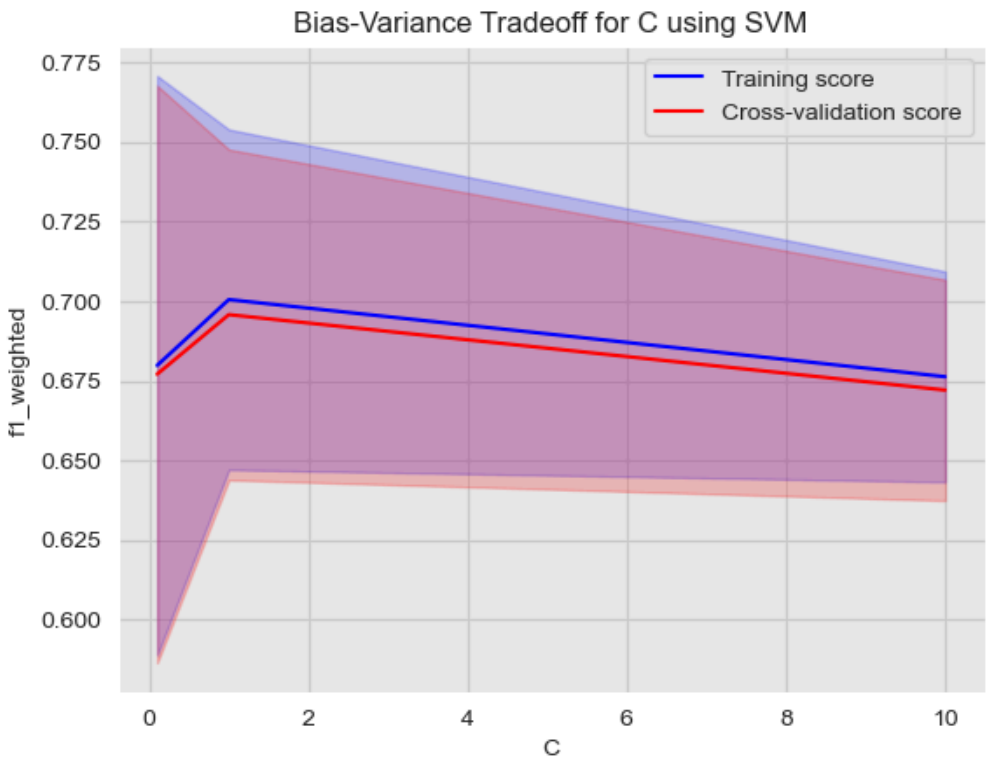


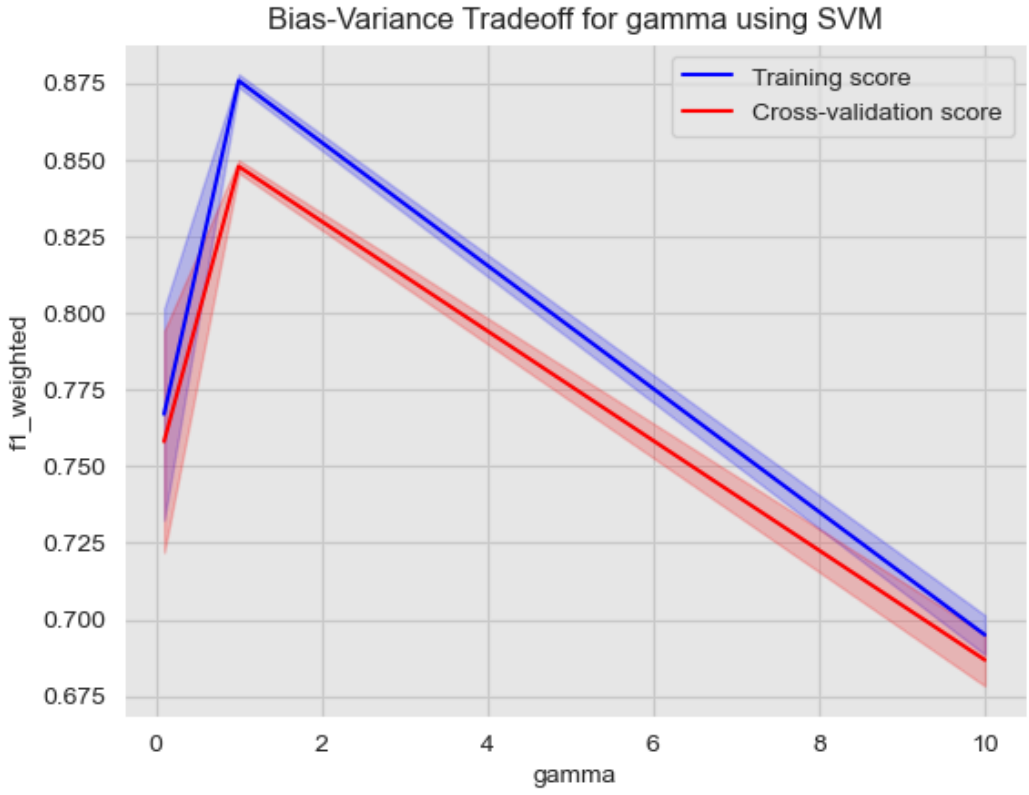
Classification report:

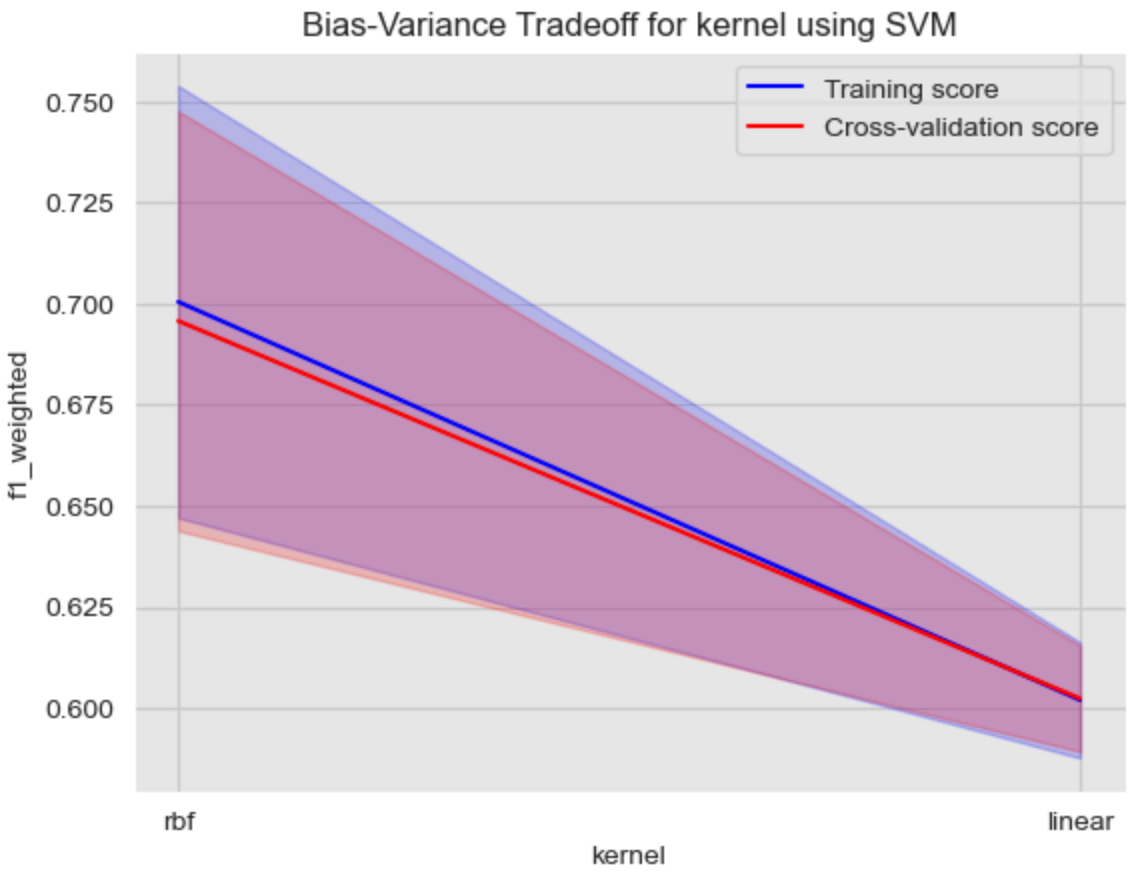
```
Classification Report of SVC(C=1, gamma=1, max_iter=1000, random_state=0) is:
              precision    recall  f1-score   support

           0       0.90      0.98      0.94     89917
           1       0.10      0.02      0.04     10083

    accuracy                           0.88    100000
   macro avg       0.50      0.50      0.49    100000
weighted avg       0.82      0.88      0.85    100000
```

Train-Validation Curve

Here are some Train-Validation Curves that we further used for the hyperparameter tuning process:


Bias-Variance Tradeoff for C using SVM
Training score
Cross-validation score
f1_weighted
0.775
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0
2
4
6
8
10
C


Bias-Variance Tradeoff for gamma using SVM
Training score
Cross-validation score
f1_weighted
0.875
0.850
0.825
0.800
0.775
0.750
0.725
0.700
0.675
0
2
4
6
8
10
gamma



Bias-variance Analysis

- mean square error: 0.1202861
- bias: 0.10426006449999999
- var: 0.016026035499999983
- Estimated Eout: 0.12028609999999997

In our problem, we can see that the model has a low bias and a low variance, which means that it is well-fitted to the data and can generalize well to new, unseen data because of the small Eout.

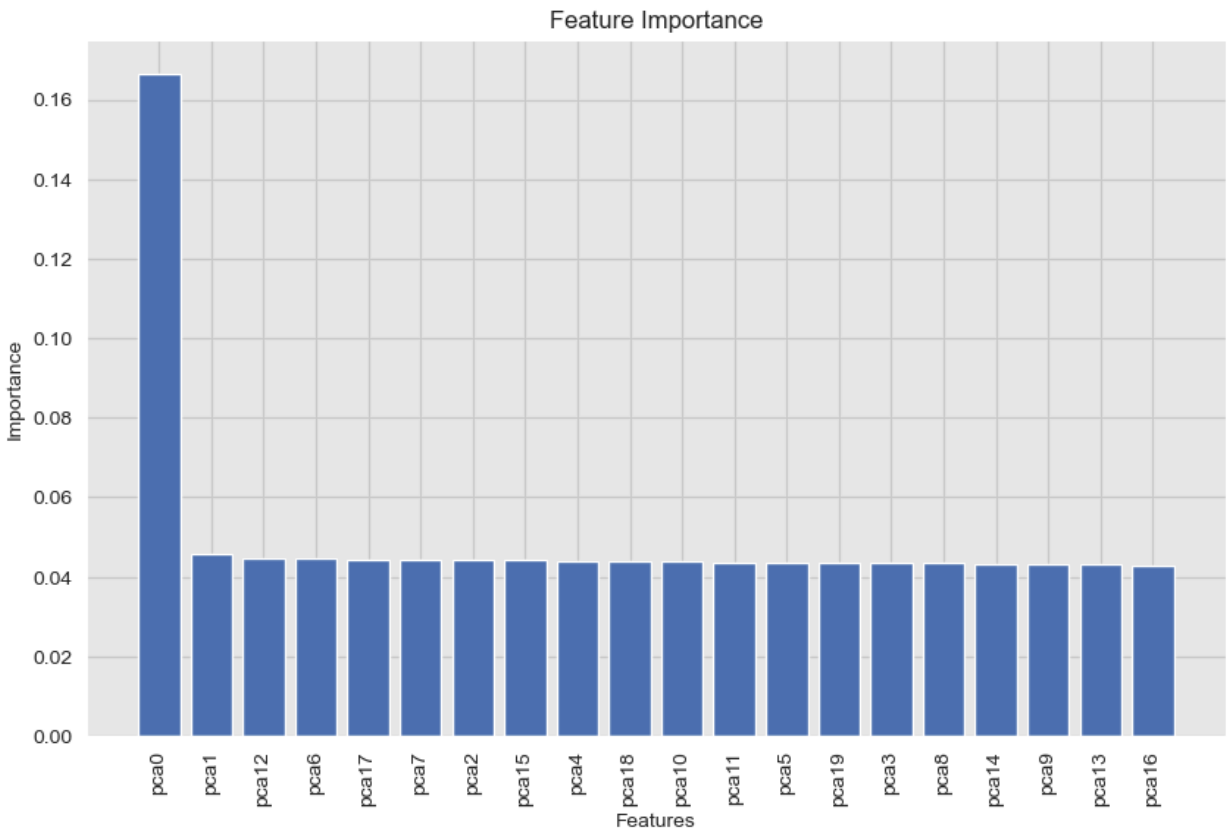
Random Forest

Random Forest is a popular machine learning algorithm that falls under the category of ensemble learning methods. It is a type of decision tree algorithm that generates multiple decision trees and combines their predictions to produce the final output.

- Advantages of Random Forest:
    - High accuracy: Random Forest has a high accuracy rate due to the combination of multiple decision trees.
    - Robustness: It is robust to outliers and noise in the dataset.
    - Feature importance: Random Forest provides a measure of feature importance, which can be useful for feature selection and interpretation.
    - Scalability: It is able to handle large datasets and can be parallelized for faster processing.
    - Low risk of overfitting: The combination of multiple decision trees reduces the risk of overfitting and increases generalization.
- Disadvantages of Random Forest:
    - Lack of interpretability: Random Forest models are often difficult to interpret due to their complexity and the number of decision trees.
    - Computationally expensive: The training and prediction process of Random Forest can be computationally expensive, especially for large datasets.
    - Memory usage: The memory usage of Random Forest can be high due to the number of decision trees.
    - Biased towards the majority class: Random Forest can be biased towards the majority class in imbalanced datasets, leading to lower accuracy for the minority class.

Feature Importance Plot

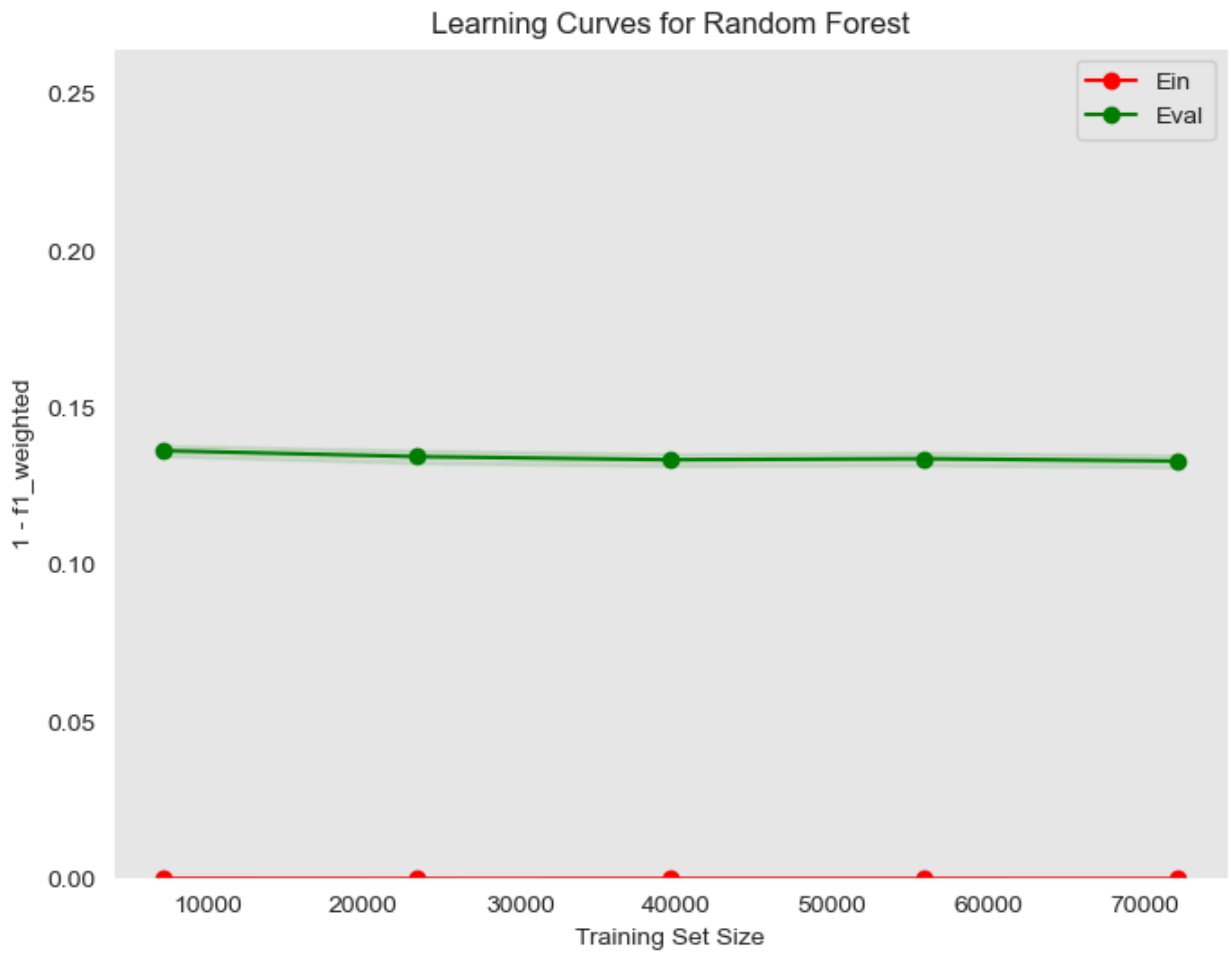
Note that we now only use 20 features that we got from PCA to be able to train and apply cross-validation on all models.
A feature importance plot shows the importance of each feature in the model. It can be used to identify the most important features and to understand the impact of each feature on the model's predictions.



We can see that the first feature is the most important one and the other features are almost equally important.

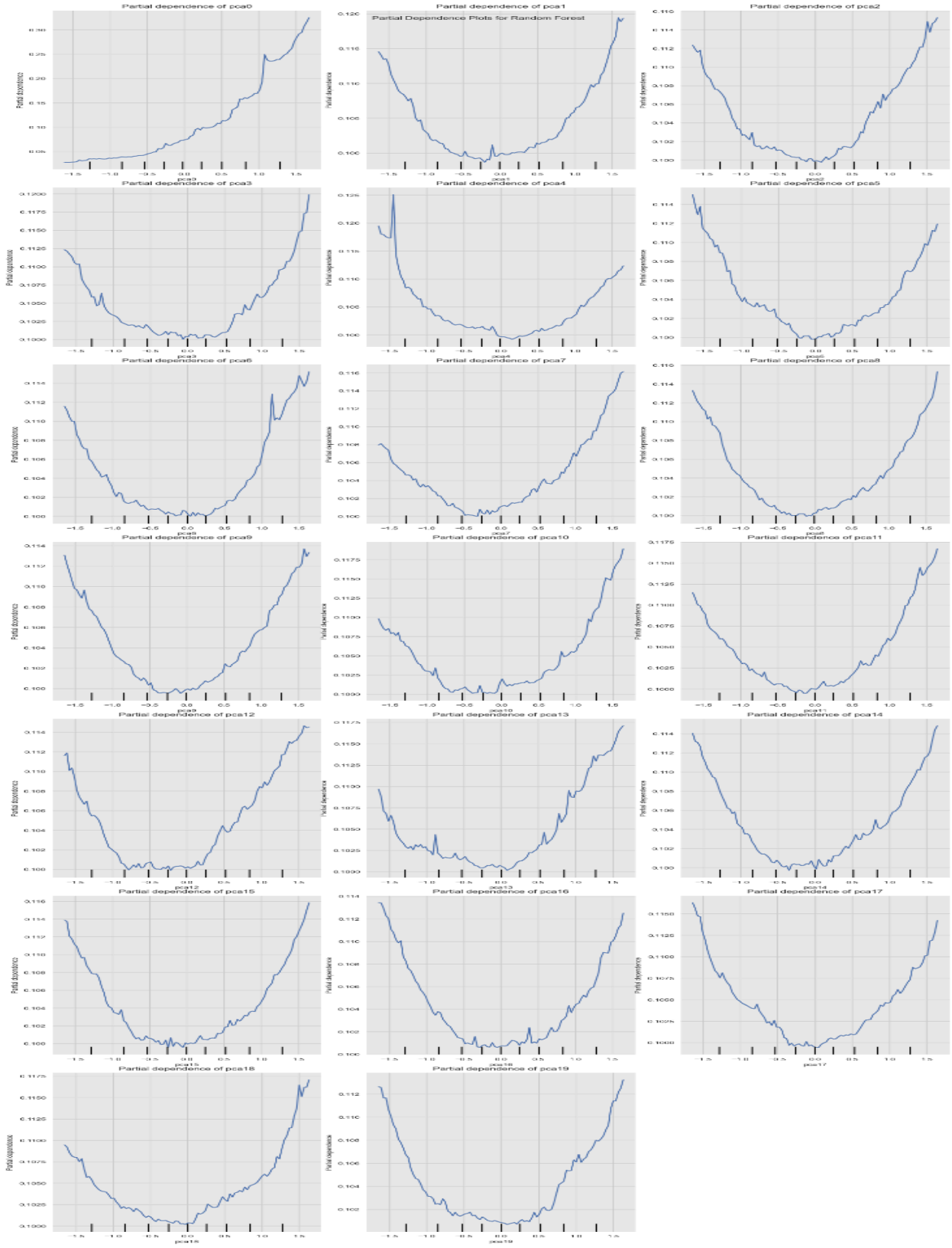
Learning Curves Plot

shows the training error (Ein) and validation error (Eval) as a function of the training set size.



Even if the error on the training is zero, there is proof that shows that random forest isn't overfitting even Ein = 0 and you can see that the error on validation isn't big even if it's far from Ein.

Partial Dependence Plot



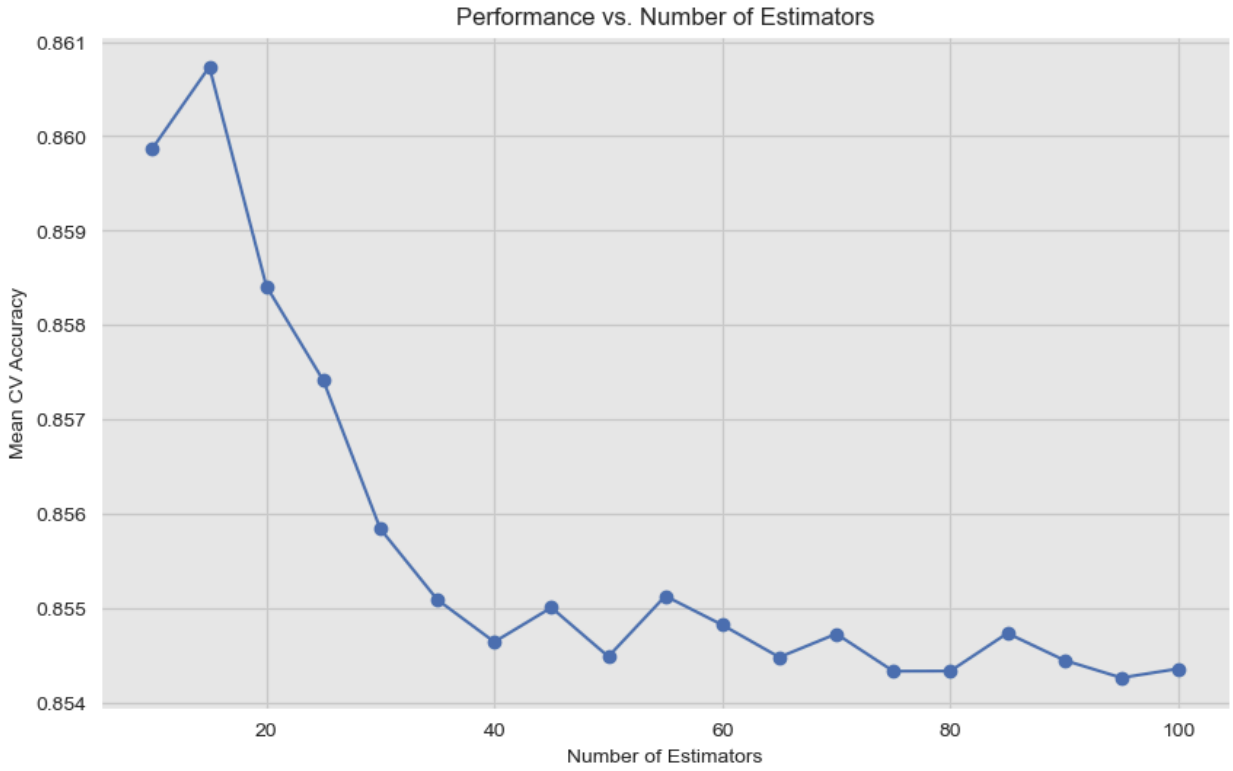
We can see that the relationship between the model and each feature is non-linear except for the first feature has a linear, directly proportional relationship with the model.

Hyperparameter Tuning

This is a process of adjusting the parameters of a model to optimize its performance. It can be done using techniques like grid search, random search, or Bayesian optimization.

Number of estimators effect

The following graph shows the effect of changing the number of estimators when the other parameters are constants.

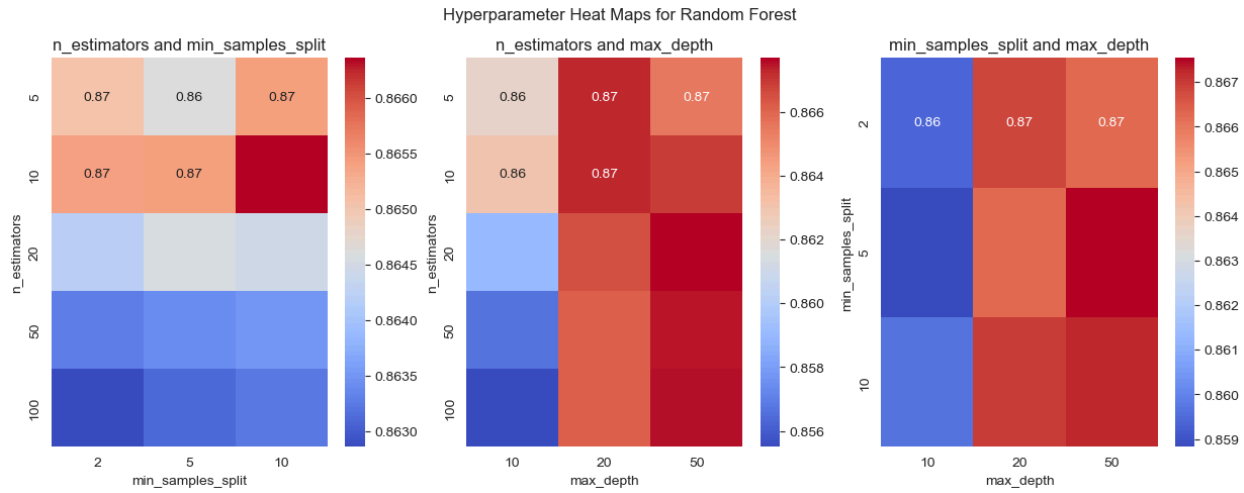


We can see that for our problem using a lot of estimators isn't a good thing because that will lead to overfitting (bad generalization).

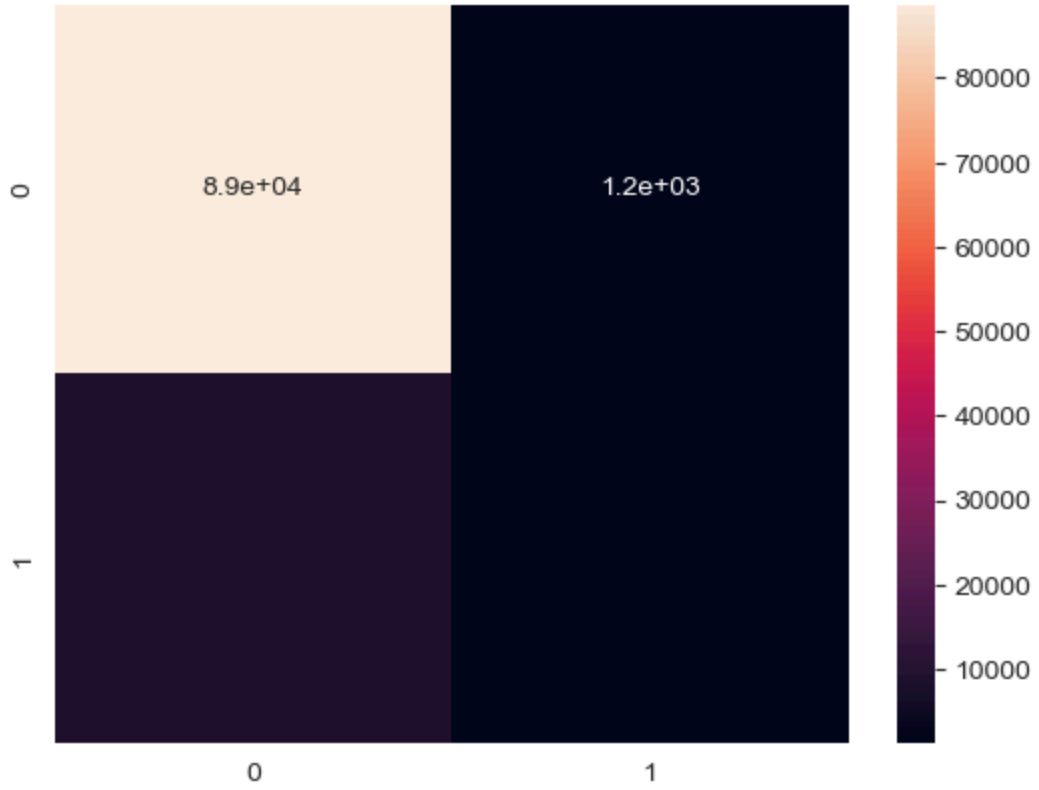
Grid Search

```
# Define the parameter grid to search over
param_grid = {
    'n_estimators': [5, 10, 20, 50, 100],
    'min_samples_split': [2, 5, 10],
    'max_depth': [10, 20, 50],
}
```

where “n_estimators” is the number of trees in the forest. Higher numbers generally reduce overfitting, but also increase computational cost. “min_samples_split” is the minimum number of samples required to split an internal node. “max_depth” is the maximum depth of the trees. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
Here are some heatmap visualizations of the grid search results:



we can see that some combinations give higher accuracy like n_estimators = 5 and min_samples_split = 10
After applying the grid search we found that the best parameters are: {'max_depth': 50, 'min_samples_split': 5, 'n_estimators': 20}
After training a model using the best parameters we got from the search grid we got the following results:
Confusion matrix:

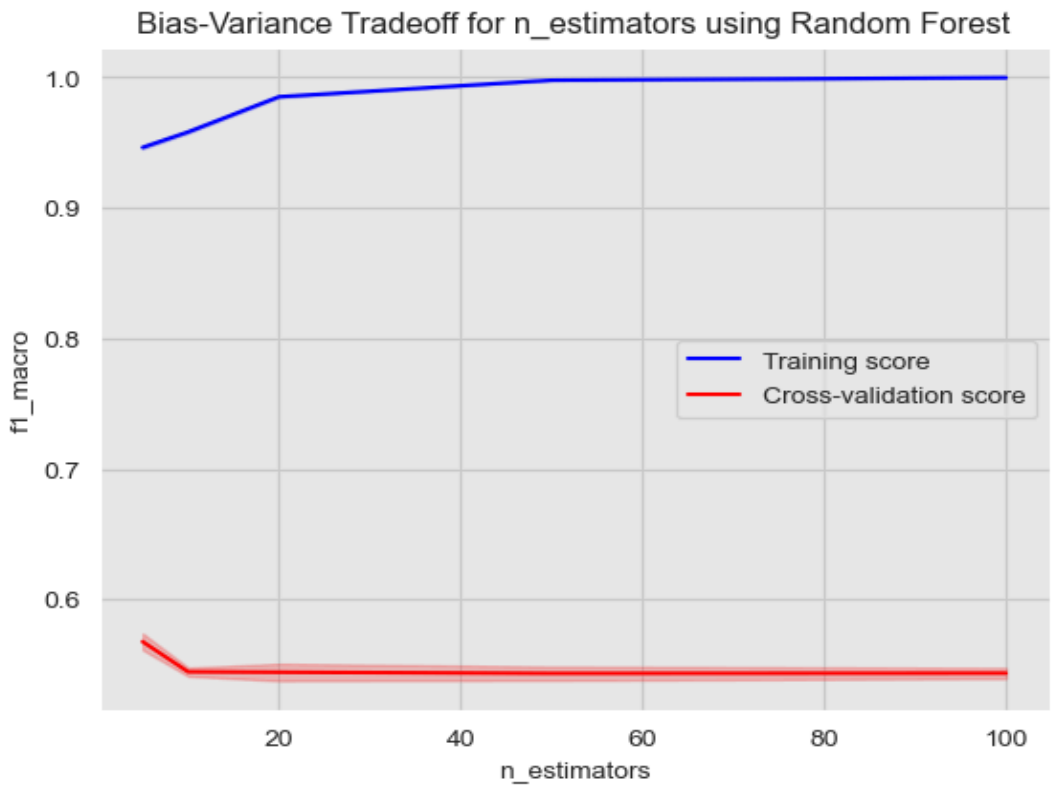


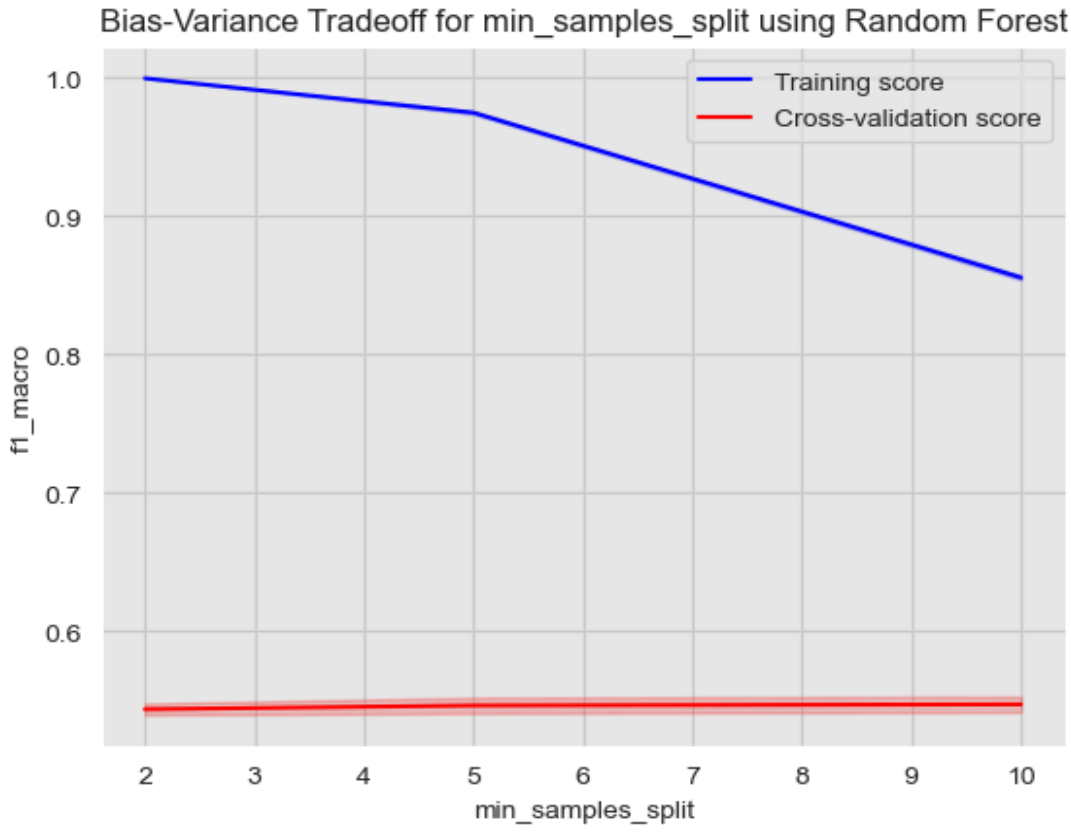
Classification report:

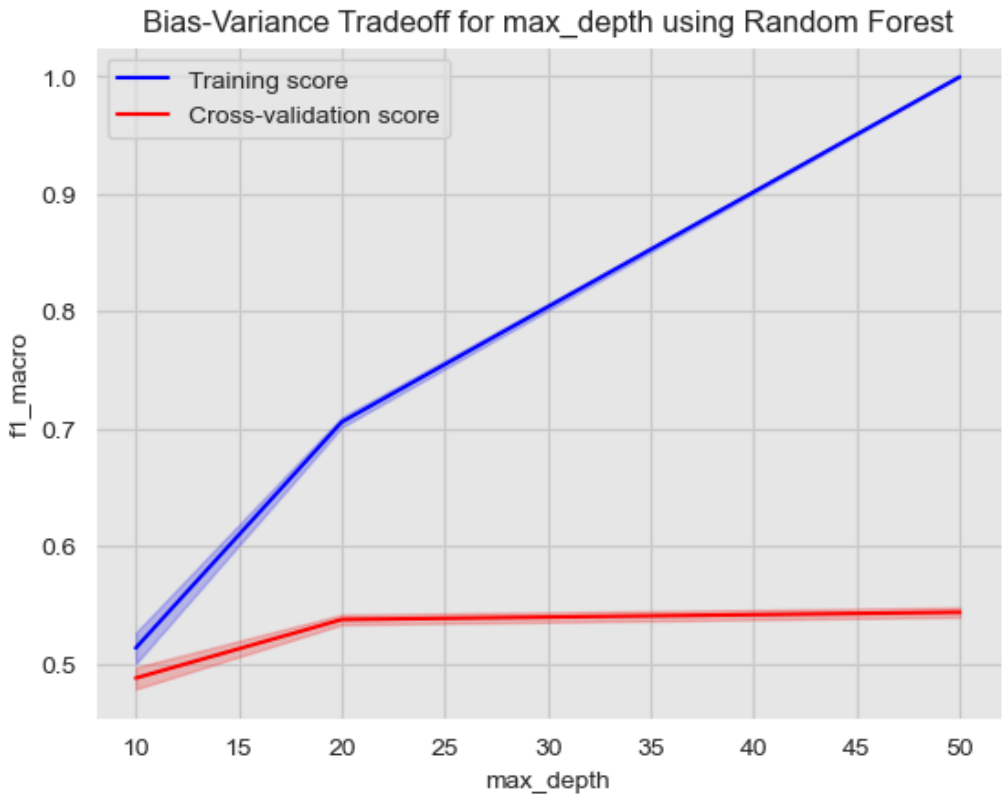
```
Classification Report of RandomForestClassifier(max_depth=50, min_samples_split=5, n_estimators=20,
                       n_jobs=-1, random_state=0) is:
              precision    recall  f1-score   support

           0       0.91      0.99      0.95     89917
           1       0.50      0.12      0.19     10083

    accuracy                           0.90    100000
   macro avg       0.71      0.55      0.57    100000
weighted avg       0.87      0.90      0.87    100000
```

Train-Validation Curve

Here are some Train-Validation Curves that we further used for the hyperparameter tuning process:


Bias-Variance Tradeoff for n_estimators using Random Forest
1.0
0.9
0.8
0.7
0.6
f1_macro
Training score
Cross-validation score
20
40
60
80
100
n_estimators


Bias-Variance Tradeoff for min_samples_split using Random Forest
1.0
0.9
0.8
0.7
0.6
f1_macro
Training score
Cross-validation score
2
3
4
5
6
7
8
9
10
min_samples_split

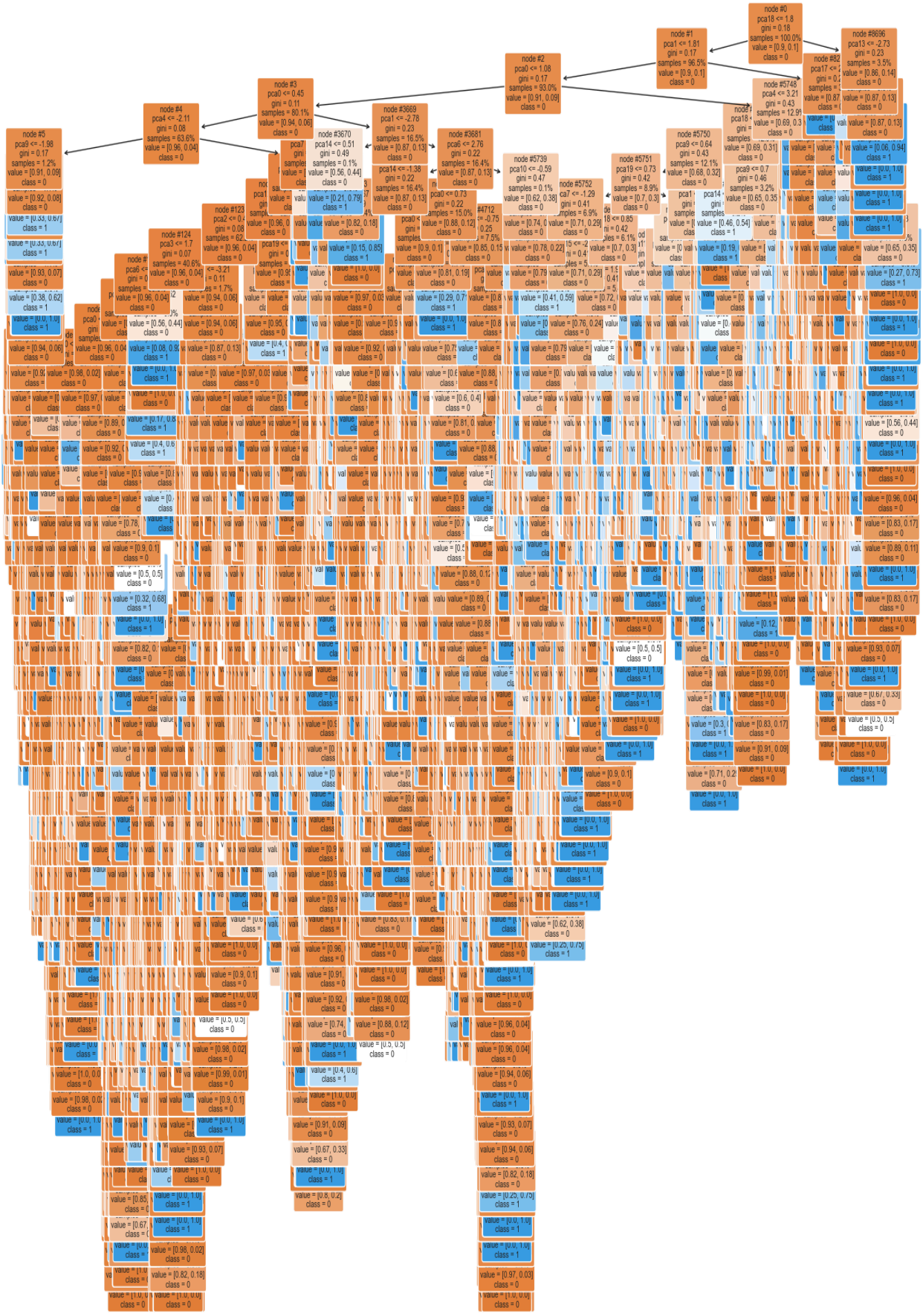


Bias-variance Analysis

- mean square error: 0.10085995
- bias: 0.08931127525000002
- var: 0.011548674750000005
- Estimated Eout: 0.10085995000000002

Tree Plot

A tree plot shows the structure of the decision trees used in the random forest. It can be used to understand how the model makes predictions.


node #0
pca18 <= 1.8
gini = 0.18
samples = 100.0%
value = [0.9, 0.1]
class = 0
node #1
pca1 <= 1.81
gini = 0.17
samples = 96.5%
value = [0.9, 0.1]
class = 0
node #2
pca0 <= 1.08
gini = 0.17
samples = 93.0%
value = [0.91, 0.09]
class = 0
node #3
pca0 <= 0.45
gini = 0.11
samples = 80.1%
value = [0.94, 0.06]
class = 0
node #4
pca4 <= -2.11
gini = 0.08
samples = 63.6%
value = [0.96, 0.04]
class = 0
node #5
pca9 <= -1.98
gini = 0.17
samples = 1.2%
value = [0.91, 0.09]
class = 0
node #3670
pca14 <= 0.51
gini = 0.49
samples = 0.1%
value = [0.56, 0.44]
class = 0
node #3669
pca1 <= -2.78
gini = 0.23
samples = 16.5%
value = [0.87, 0.13]
class = 0
node #3681
pca6 <= 2.76
gini = 0.22
samples = 16.4%
value = [0.87, 0.13]
class = 0
node #5748
pca4 <= 3.21
gini = 0.43
samples = 12.9%
value = [0.69, 0.3
class = 0
node #8696
pca13 <= -2.73
gini = 0.23
samples = 3.5%
value = [0.86, 0.14]
class = 0
node #5739
pca10 <= -0.59
gini = 0.47
samples = 0.1%
value = [0.62, 0.38]
class = 0
node #5751
pca19 <= 0.73
gini = 0.42
samples = 8.9%
value = [0.7, 0.3]
class = 0
node #5752
pca7 <= -1.29
gini = 0.41
samples = 6.9%
value = [0.71, 0.29]
class = 0
node #5750
pca9 <= 0.64
gini = 0.43
samples = 12.1%
value = [0.68, 0.32]
class = 0
node #123
pca2 <= 0.4
gini = 0.08
samples = 62
value = [0.96, 0.04]
class = 0
node #124
pca3 <= 1.7
gini = 0.07
samples = 40.6%
value = [0.96, 0.04]
class = 0
node #4712
pca0 <= -0.75
gini = 0.25
samples = 7.5%
value = [0.85, 0.15]
class = 0
node #
pca18 <=
gini =
samples
value =
class =
node #
pca17 <=
gini =
samples
value =
class =
node #
pca1 <=
gini =
samples
value =
class =
node #
pca14 <=
gini =
samples
value =
class =
node #
pca0 <=
gini =
samples
value =
class =
node #
pca19 <=
gini =
samples
value =
class =
node #
pca6 <=
gini =
samples
value =
class =
node #
pca0 <=
gini =
samples
value =
class =

Variable importance
The plot can show which variables (or features) are the most important in making the predictions. The importance of a feature is determined by how much the tree nodes that use that feature reduce impurity (i.e., increase homogeneity). In our case the most important feature is pca0
Interactions between features
The plot can show how different features interact with each other to make predictions. For example, if two features are highly correlated, the plot can show whether the random forest is consistently using one feature over the other or if it's using both in combination.
Overfitting
In our case, the model isn't overfitted because there are not many shallow trees as if there are many shallow trees (i.e., with few splits) in the forest, indicating that the model is not capturing the underlying patterns in the data. This means that the model is capturing the underlying patterns in the data and has good generalization.

Adaboost

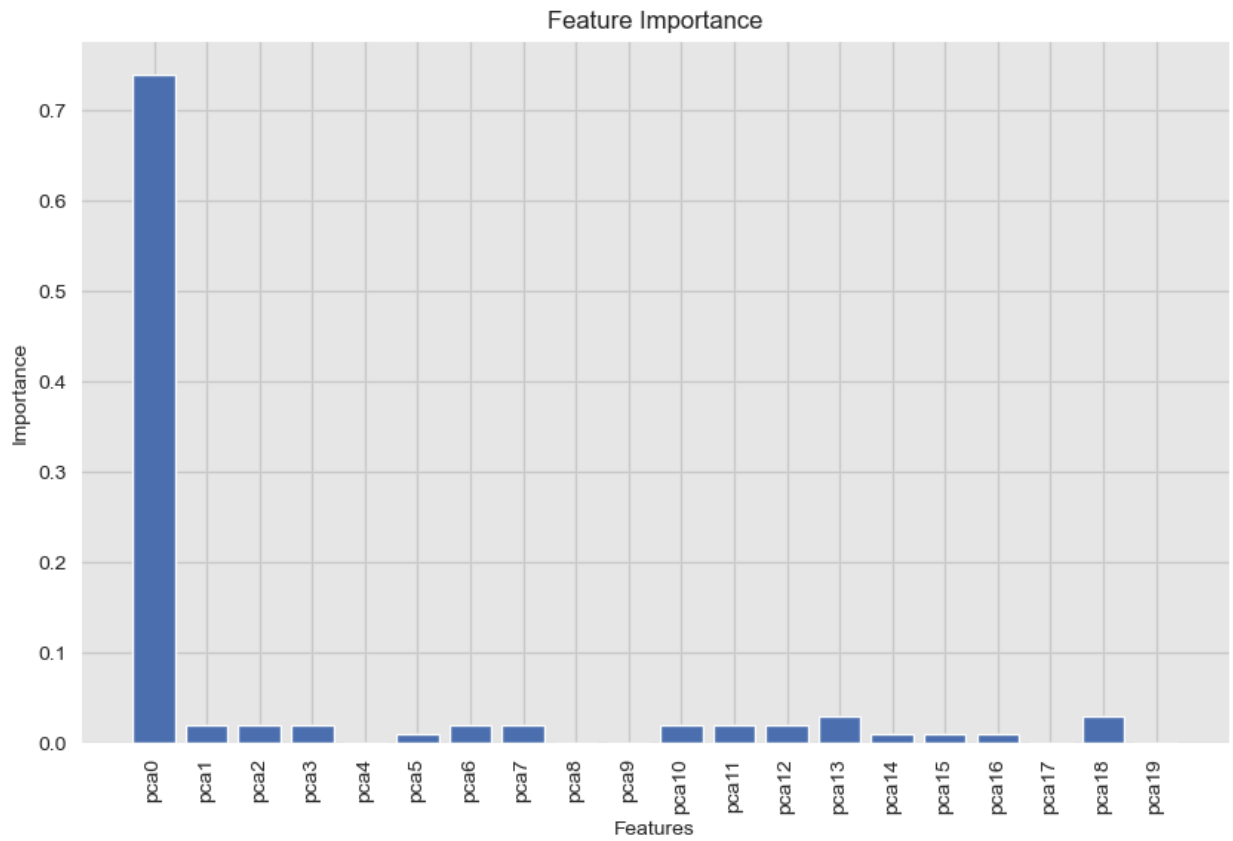
AdaBoost, short for Adaptive Boosting, is a popular ensemble learning algorithm that combines multiple weak learners to create a strong classifier. It operates by sequentially training a series of weak learners, such as decision trees with limited depth, and assigns higher weights to misclassified instances in each iteration. This iterative process focuses on the difficult instances, gradually improving the model's performance.

- **Advantages:**
    - High Accuracy: AdaBoost often achieves higher accuracy compared to individual weak learners, making it suitable for a wide range of classification tasks.
    - Versatility: It can be used with various base classifiers, such as decision trees, to build a strong ensemble model, providing flexibility in model selection.
    - Feature Selection: AdaBoost naturally selects important features by assigning higher weights to them during training, which can improve model interpretability and reduce overfitting.

- **Disadvantages:**
    - Sensitive to Noisy Data: AdaBoost is sensitive to noisy data and outliers, which can negatively impact model performance, especially when the base classifiers are affected by noise.
    - Computationally Expensive: The sequential nature of AdaBoost training makes it computationally expensive, particularly when using a large number of iterations or complex base classifiers.
    - Requires Tuning: AdaBoost requires careful tuning of hyperparameters, such as the number of estimators and learning rate, to achieve optimal performance, which can be time-consuming and resource-intensive.

Feature Importance Plot

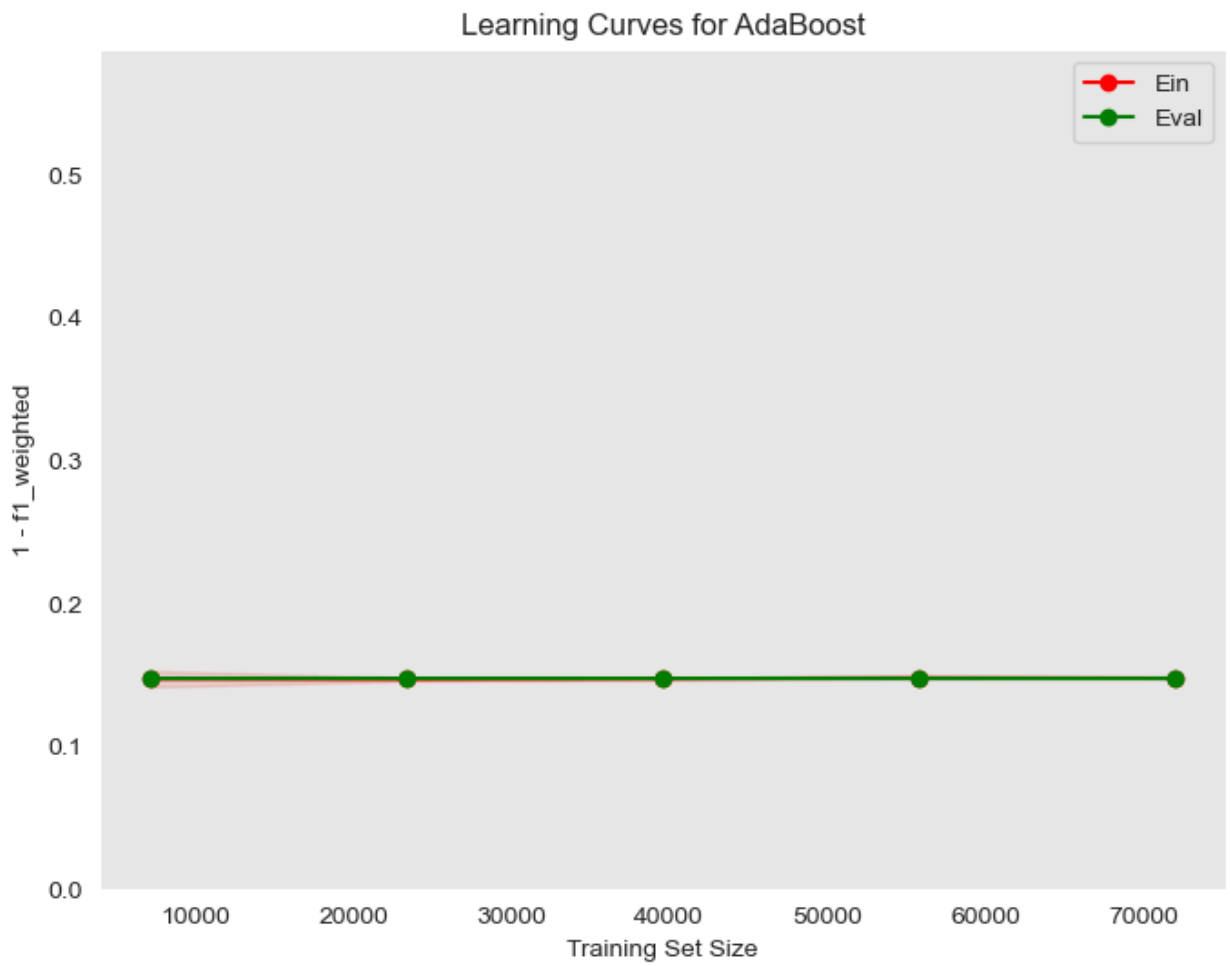
Note that we now only use 20 features that we got from PCA to be able to train and apply cross-validation on all models.
A feature importance plot shows the importance of each feature in the model. It can be used to identify the most important features and to understand the impact of each feature on the model's predictions.



We can see that the first feature is the most important one and the other features are almost equally important except for pca4, pca8, pca9, and pca17 which have zero importance.

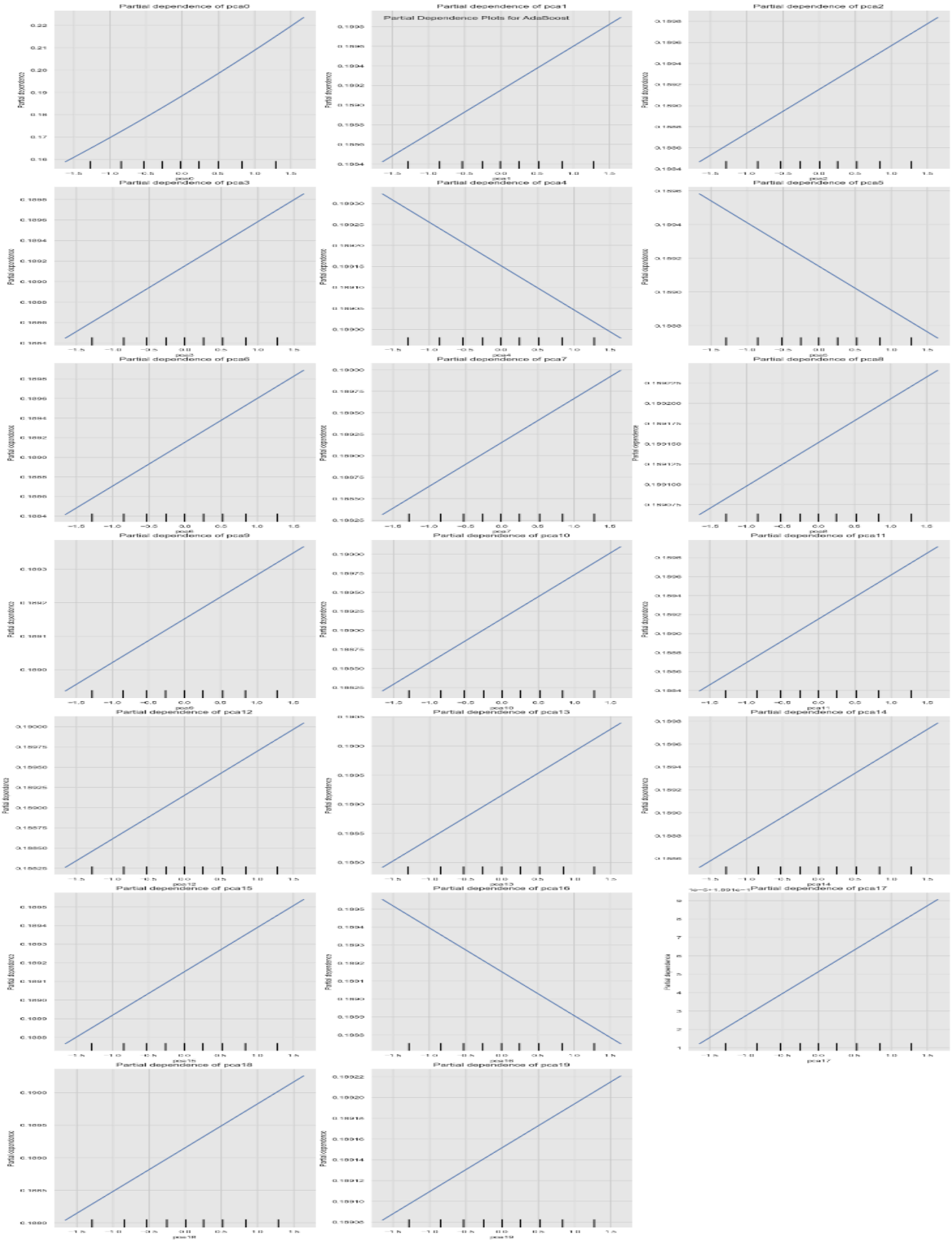
Learning Curves Plot

shows the training error (Ein) and validation error (Eval) as a function of the training set size.



We can see that the training error is near to the validation error so the model can generalize pretty well.

Partial Dependence Plot



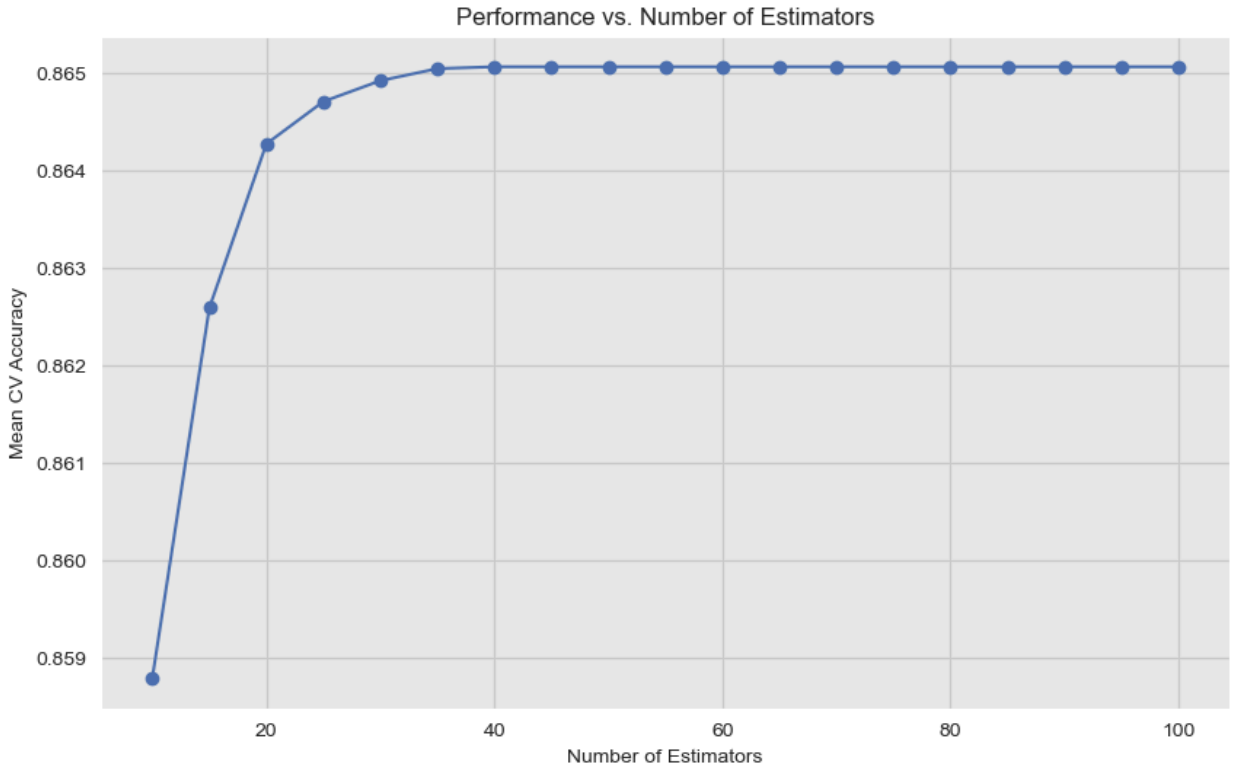
We can see that all features have a linear relationship either direct or inversely proportional.

Hyperparameter Tuning

This is a process of adjusting the parameters of a model to optimize its performance. It can be done using techniques like grid search, random search, or Bayesian optimization.

Number of estimators effect

The following graph shows the effect of changing the number of estimators when the other parameters are constants.

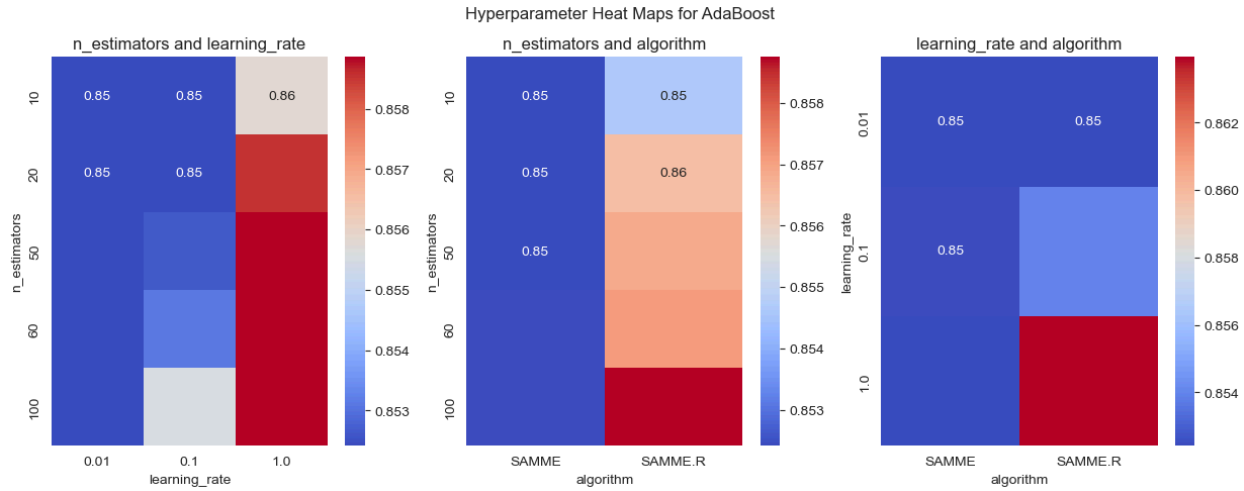


We can see that for our problem using a lot of estimators isn't a good thing because that will increase the computations even, though there is no increase in the accuracy after 40 estimators.

Grid Search

```
# Define the parameter grid to search over
param_grid = {
    'n_estimators': [10,20,50,60,100],
    'learning_rate': [0.01,0.1,1],
    'algorithm': ['SAMME', 'SAMME.R']
}
```

where “n_estimators” is the maximum number of estimators at which boosting is terminated. In other words, the number of boosting rounds (or weak learners) to train. “learning_rate” is the shrinkage parameter that controls the contribution of each classifier. Lower values require more estimators. “algorithm” is the algorithm to use for the boosting process. It can be either 'SAMME' or 'SAMME.R'.
Here are some heatmap visualizations of the grid search results:

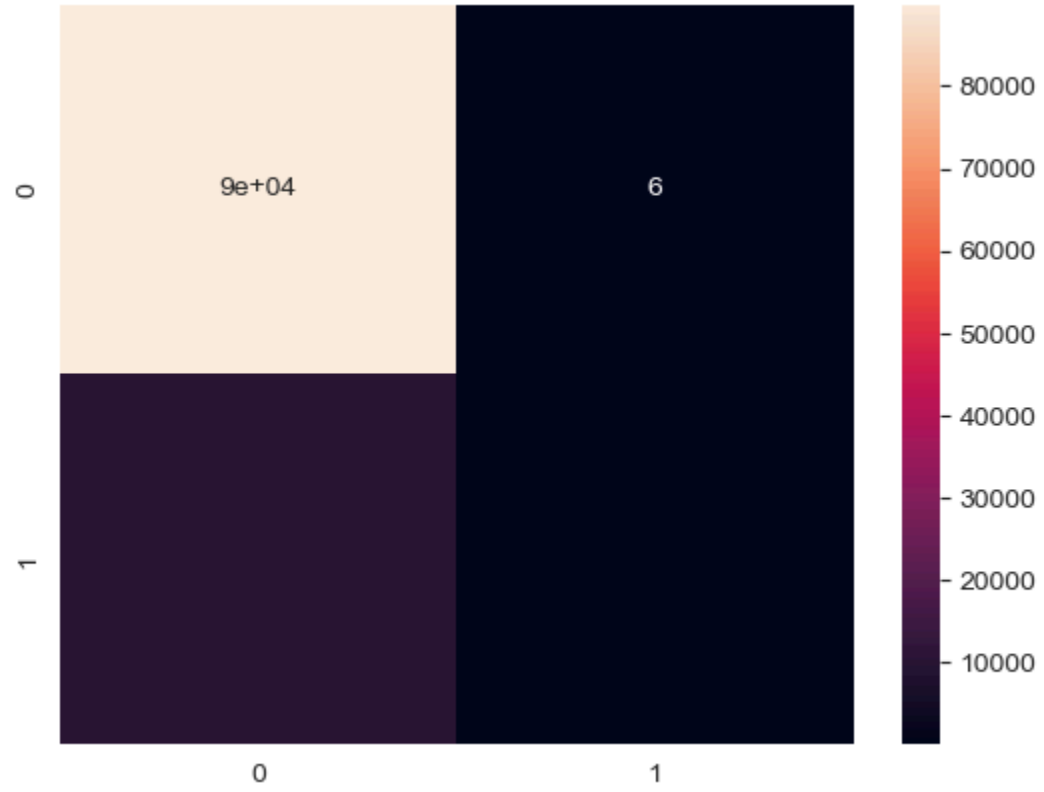


we can see that some combinations give higher accuracy like n_estimators = 10 and algorithm= SAMME.R

After applying the grid search we found that the best parameters are: {'algorithm': 'SAMME.R', 'learning_rate': 1, 'n_estimators': 50}

After training a model using the best parameters we got from the search grid we got the following results:

Confusion matrix:

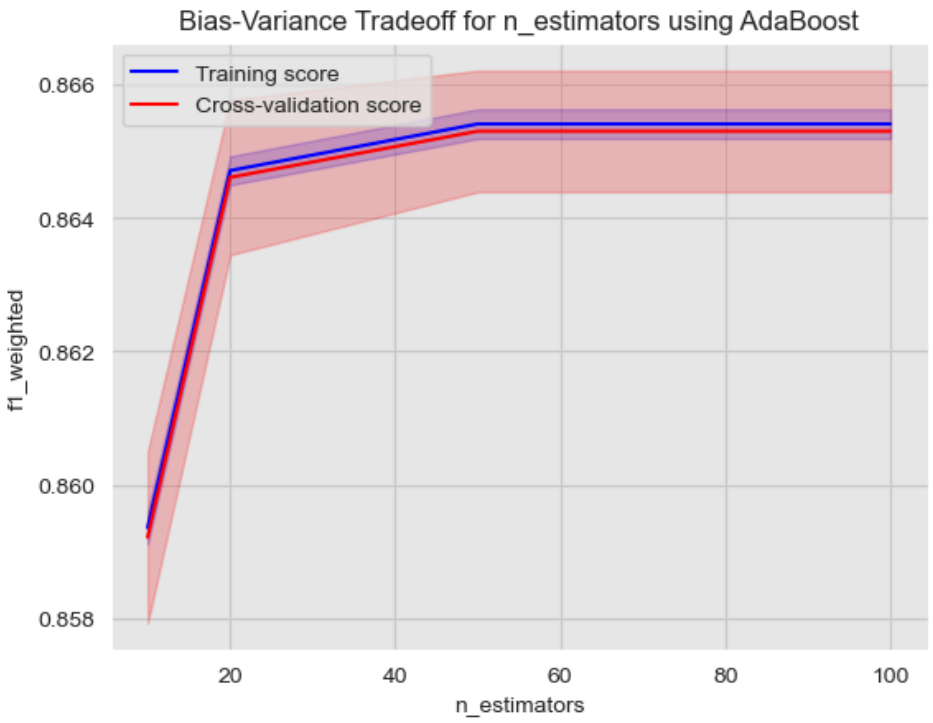


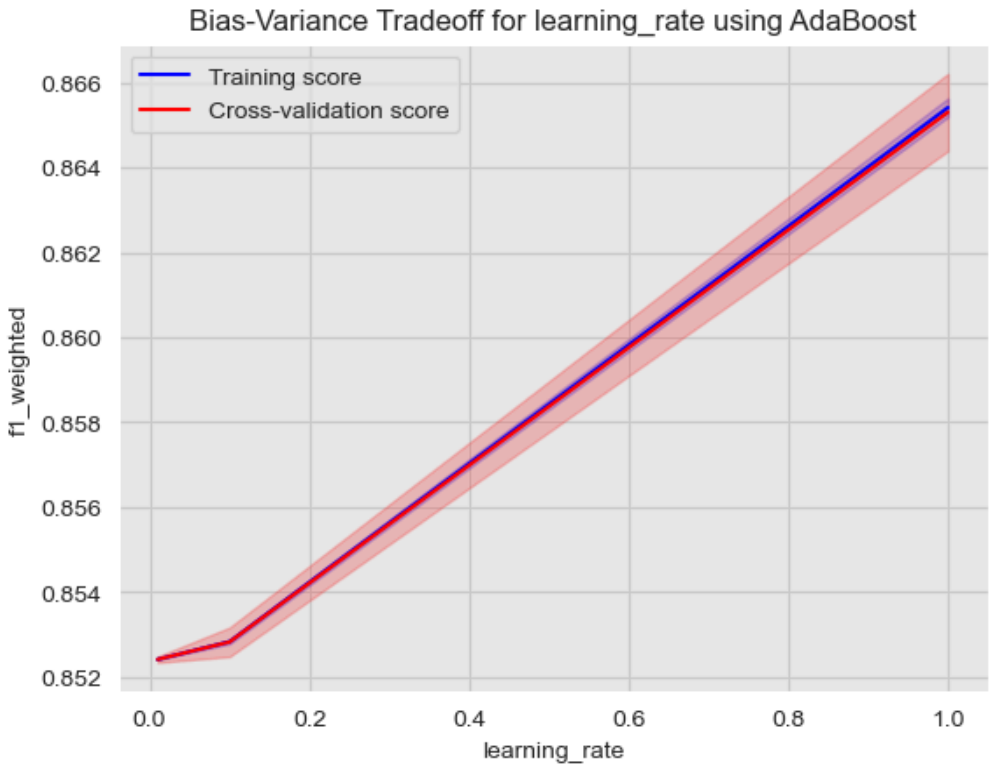
Classification report:

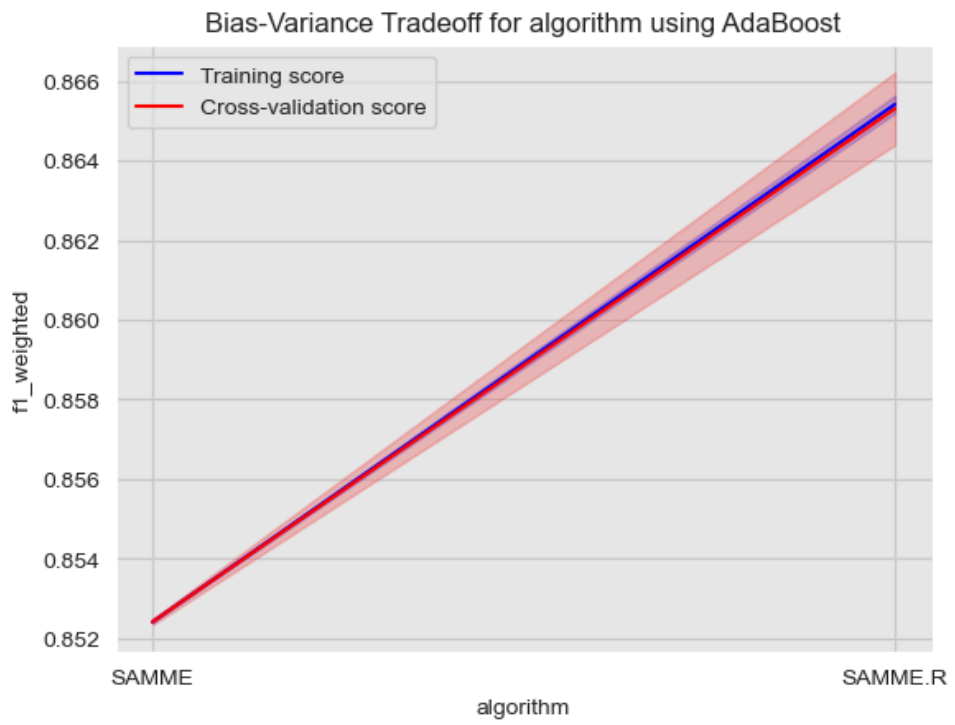
```
Classification Report of AdaBoostClassifier(estimator=LogisticRegression(max_iter=1000, random_state=42),
                   learning_rate=0.1, random_state=42) is:
              precision    recall  f1-score   support

           0       0.90      1.00      0.95     89917
           1       0.85      0.00      0.01     10083

    accuracy                           0.90    100000
   macro avg       0.88      0.50      0.48    100000
weighted avg       0.89      0.90      0.85    100000
```

Train-Validation Curve

Here are some Train-Validation Curves that we further used for the hyperparameter tuning process:


Bias-Variance Tradeoff for n_estimators using AdaBoost
Training score
Cross-validation score
0.866
0.864
0.862
0.860
0.858
f1_weighted
20
40
60
80
100
n_estimators


Bias-Variance Tradeoff for learning_rate using AdaBoost
Training score
Cross-validation score
0.866
0.864
0.862
0.860
0.858
0.856
0.854
0.852
f1_weighted
0.0
0.2
0.4
0.6
0.8
1.0
learning_rate



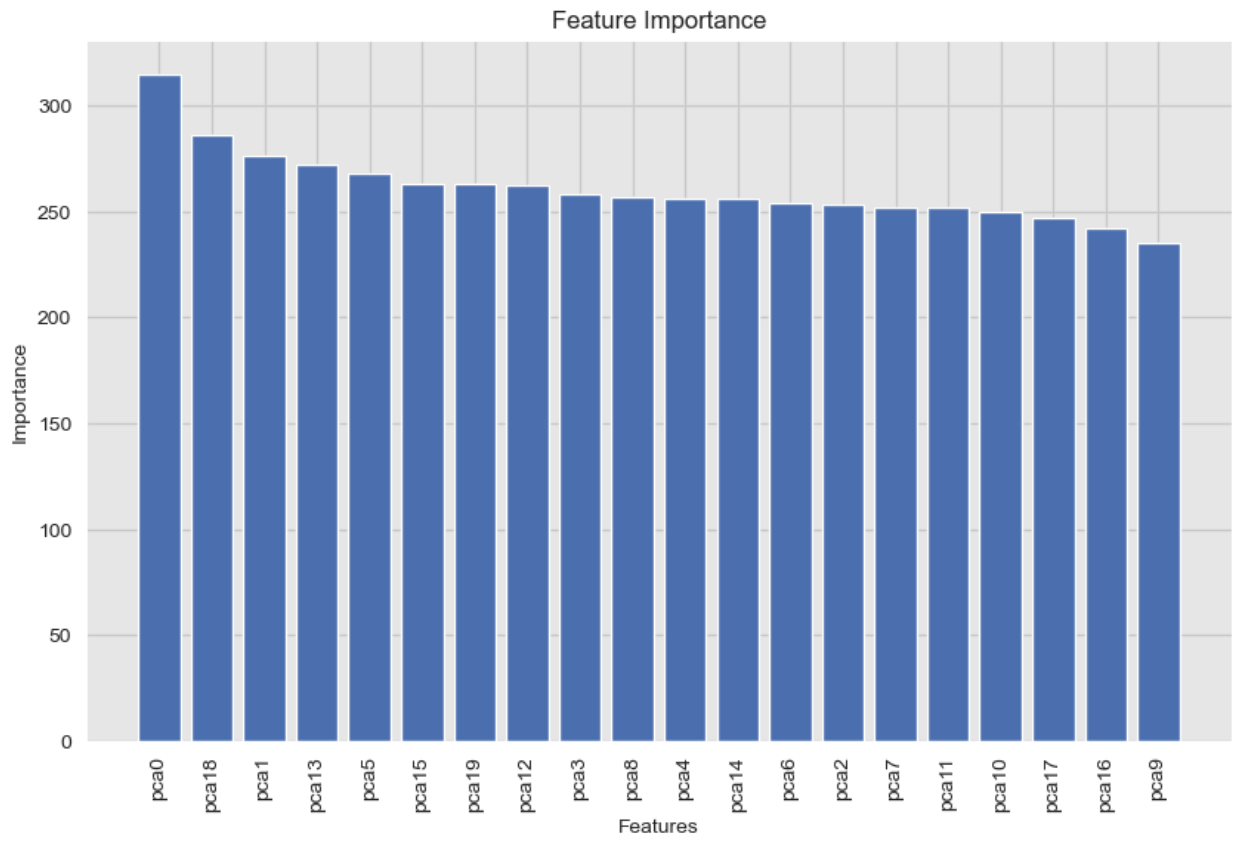
Bias-variance Analysis

- mean square error: 0.1005537
- bias: 0.10051915900000002
- var: 3.454099999999998e-05
- Estimated Eout: 0.10055370000000002

Decision tree boosting (XGboost)

Feature Importance Plot

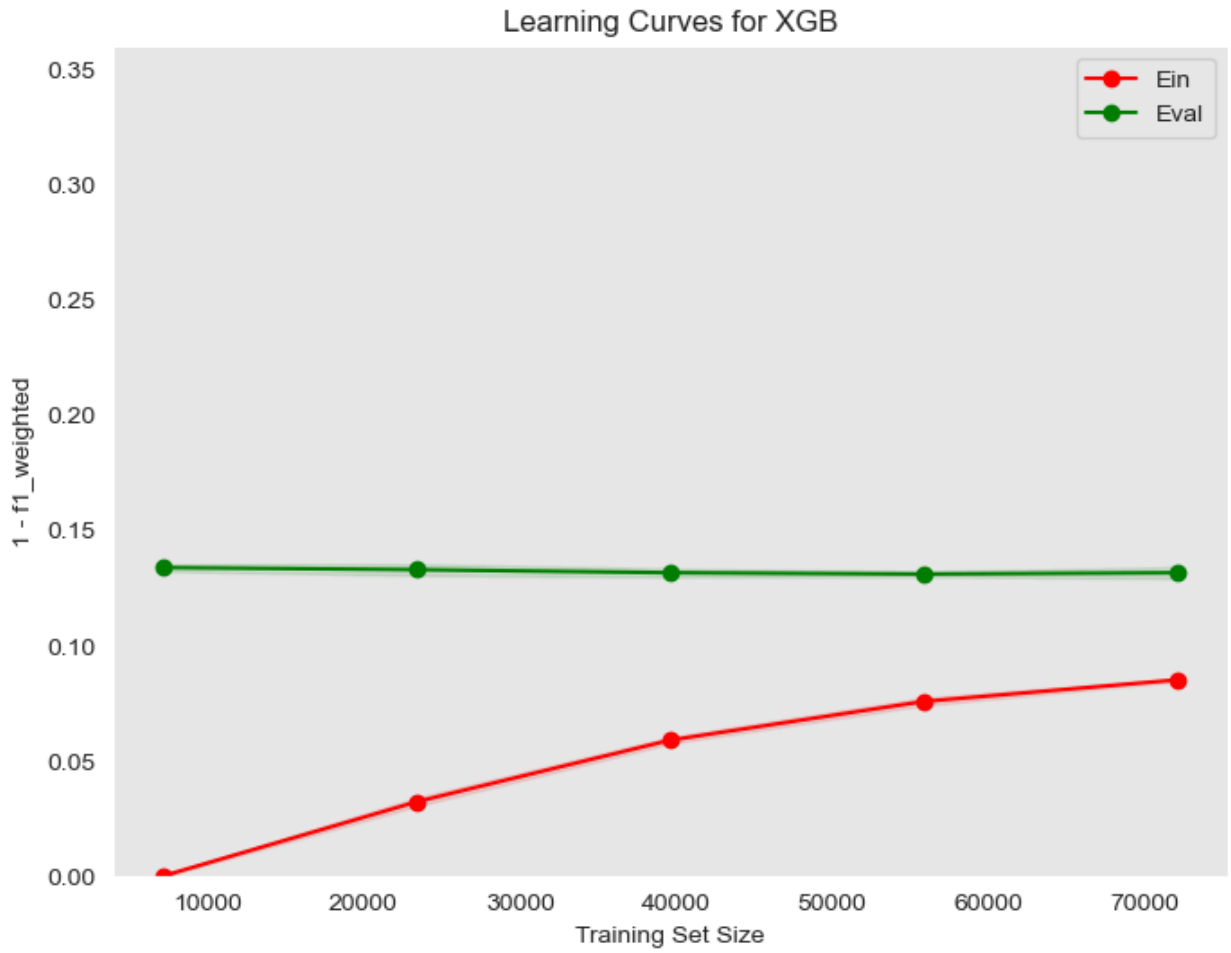
Note that we now only use 20 features that we got from PCA to be able to train and apply cross-validation on all models.
A feature importance plot shows the importance of each feature in the model. It can be used to identify the most important features and to understand the impact of each feature on the model's predictions.



We can see that the first feature is the most important one and the graph shows that XGB makes full use of all features.

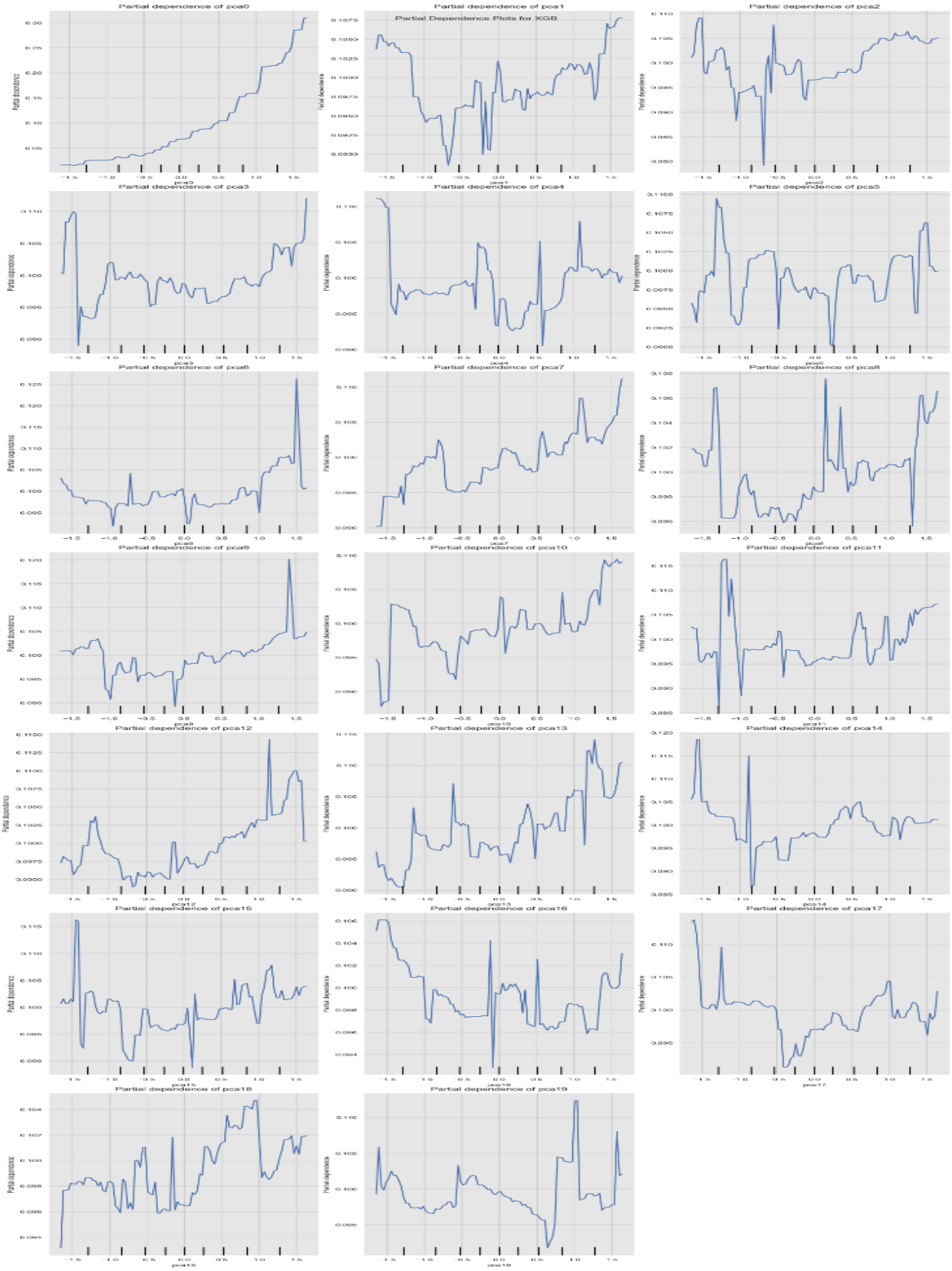
Learning Curves Plot

shows the training error (Ein) and validation error (Eval) as a function of the training set size.



We can see that with the increase in training size, the difference between Ein and Eval decreases which means a better generalization.

Partial Dependence Plot



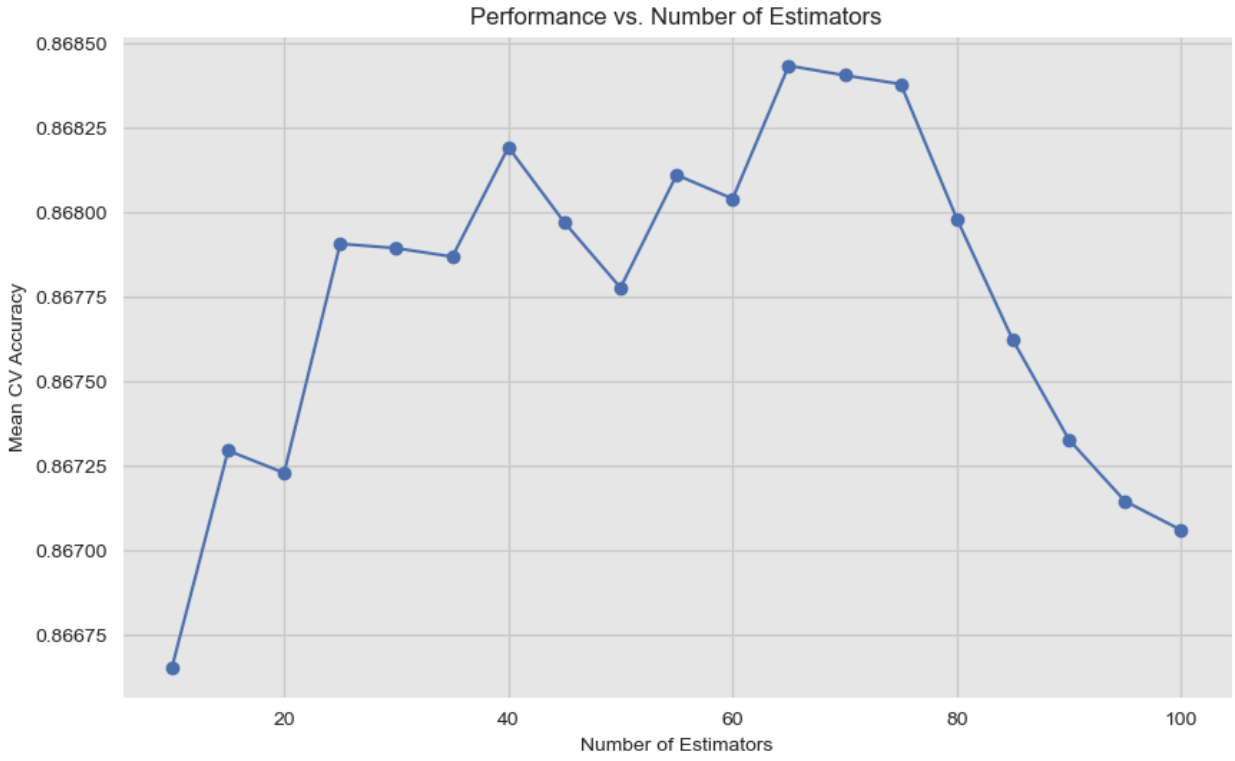
We can see that all features have a non-linear relationship either direct or inversely proportional.

Hyperparameter Tuning

This is a process of adjusting the parameters of a model to optimize its performance. It can be done using techniques like grid search, random search, or Bayesian optimization.

Number of estimators effect

The following graph shows the effect of changing the number of estimators when the other parameters are constants.



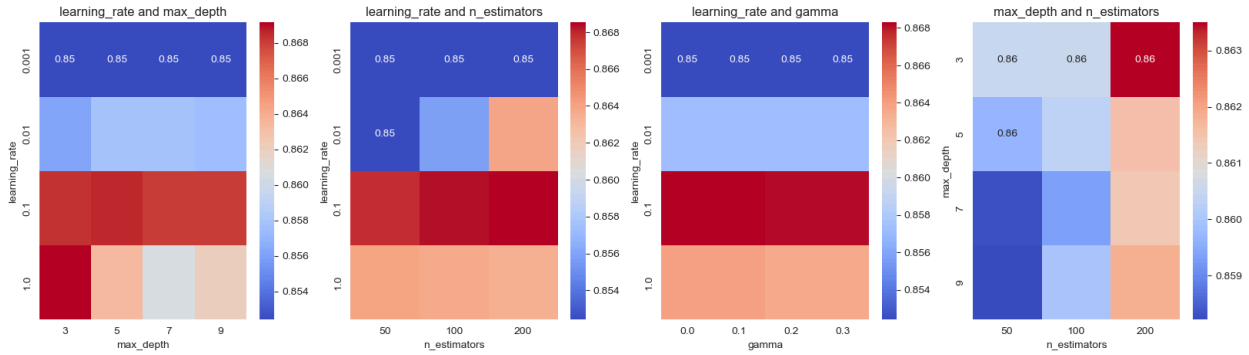
We can see that for our problem using a lot of estimators isn't a good thing because after 70 estimators the accuracy decreases.
Also, notice that at a number of estimators = 50, it's less than 40 estimators so we need the number of estimators carefully based on the problem.

Grid Search

```
param_grid = {
    'learning_rate': [0.001, 0.01, 0.1, 1],
    'max_depth': [3, 5, 7, 9],
    'n_estimators': [50, 100, 200],
    'gamma': [0, 0.1, 0.2, 0.3],
}
```

where “n_estimators” is the maximum number of estimators at which boosting is terminated. In other words, the number of boosting rounds (or weak learners) to train. “learning_rate” is the shrinkage parameter that controls the contribution of each classifier. Lower values require more estimators. “gamma” is the minimum loss reduction required to make a further partition on a leaf node of the tree. Higher values make the algorithm more conservative. “max_depth” is the maximum depth of a tree. Increasing it makes the model more complex and more likely to overfit. 0 indicates no limit on depth.

Here are some heatmap visualizations of the grid search results:

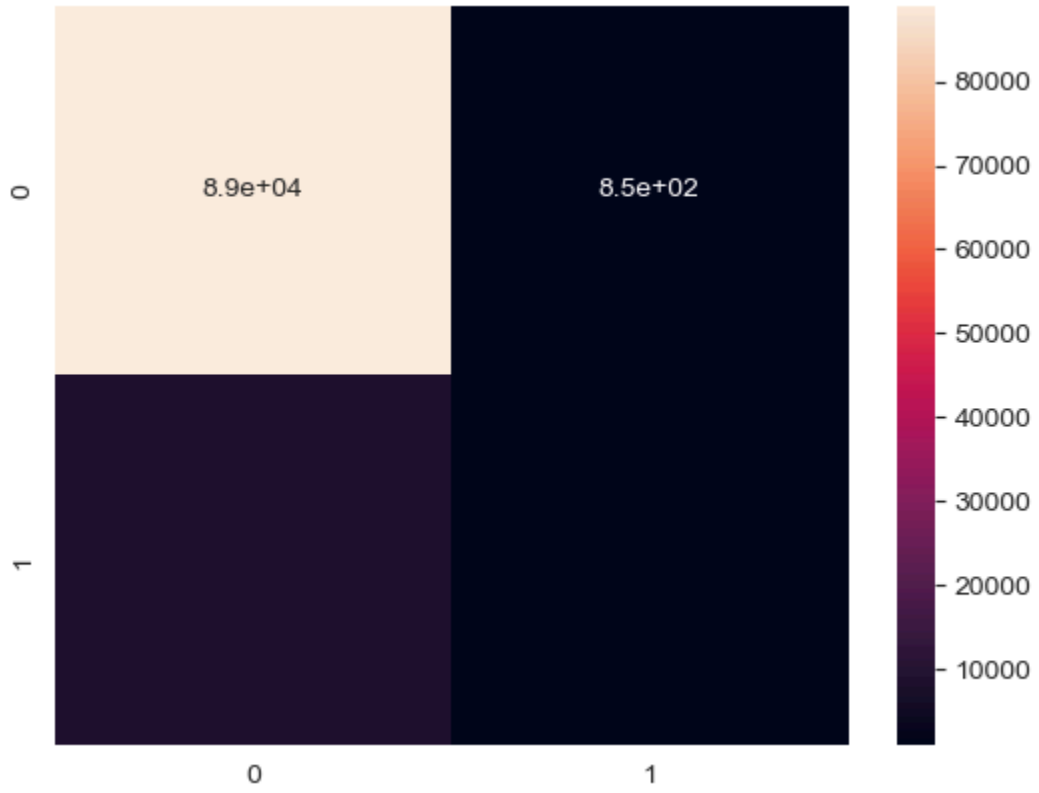


we can see that some combinations give higher accuracy like n_estimators = 200 and learning rare = 0.85

After applying the grid search we found that the best parameters are:
{'gamma': 0.1, 'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200}

After training a model using the best parameters we got from the search grid we got the following results:

Confusion matrix:

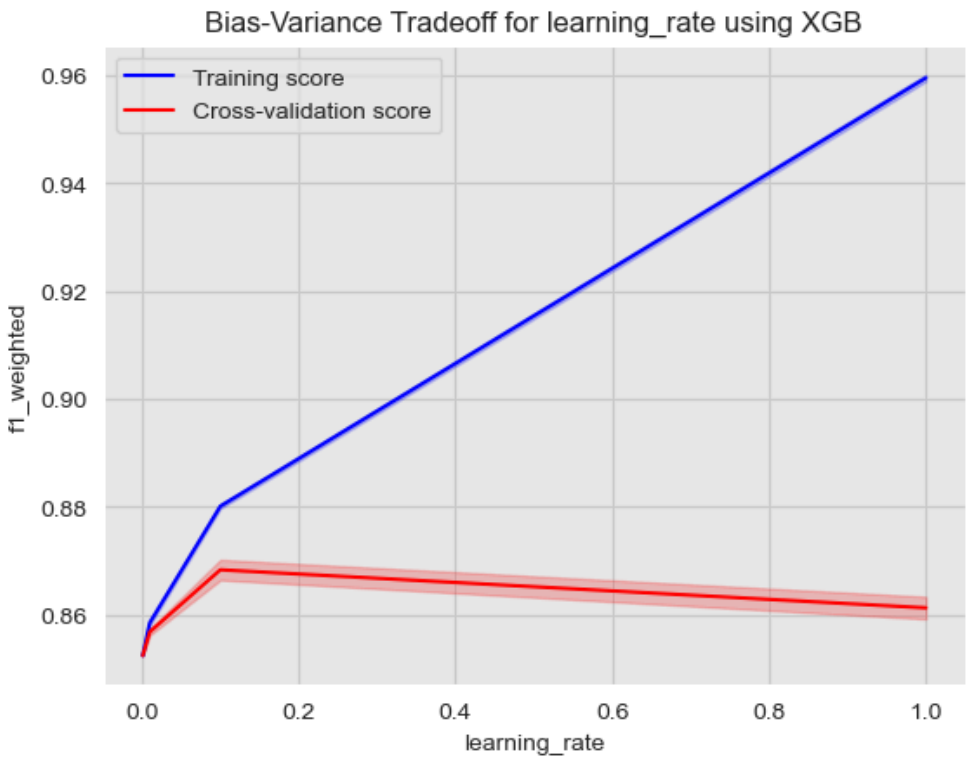


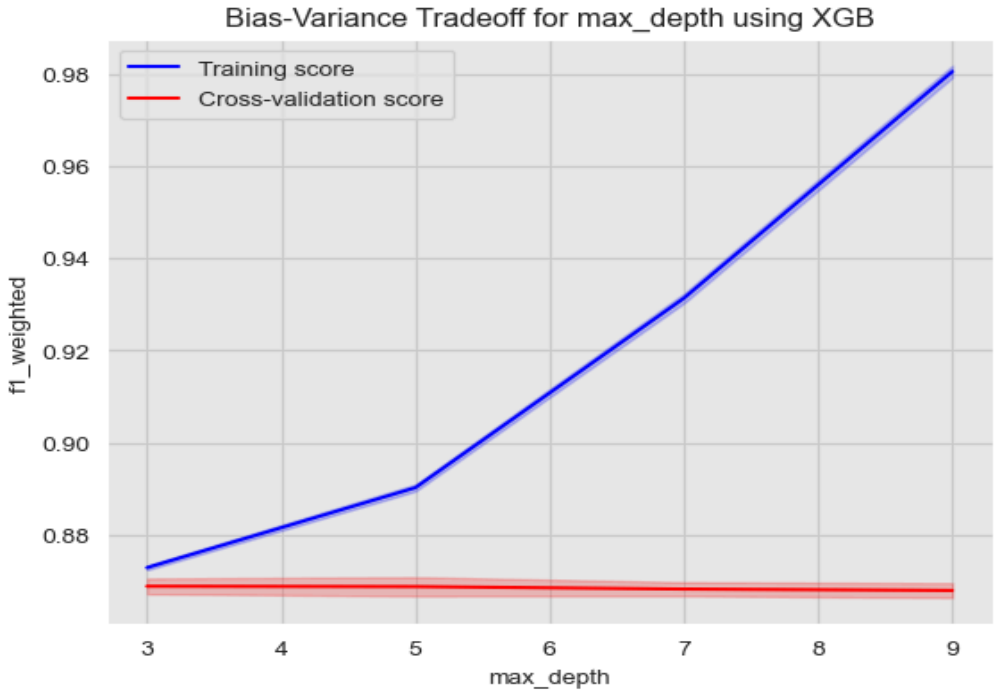
Classification report:

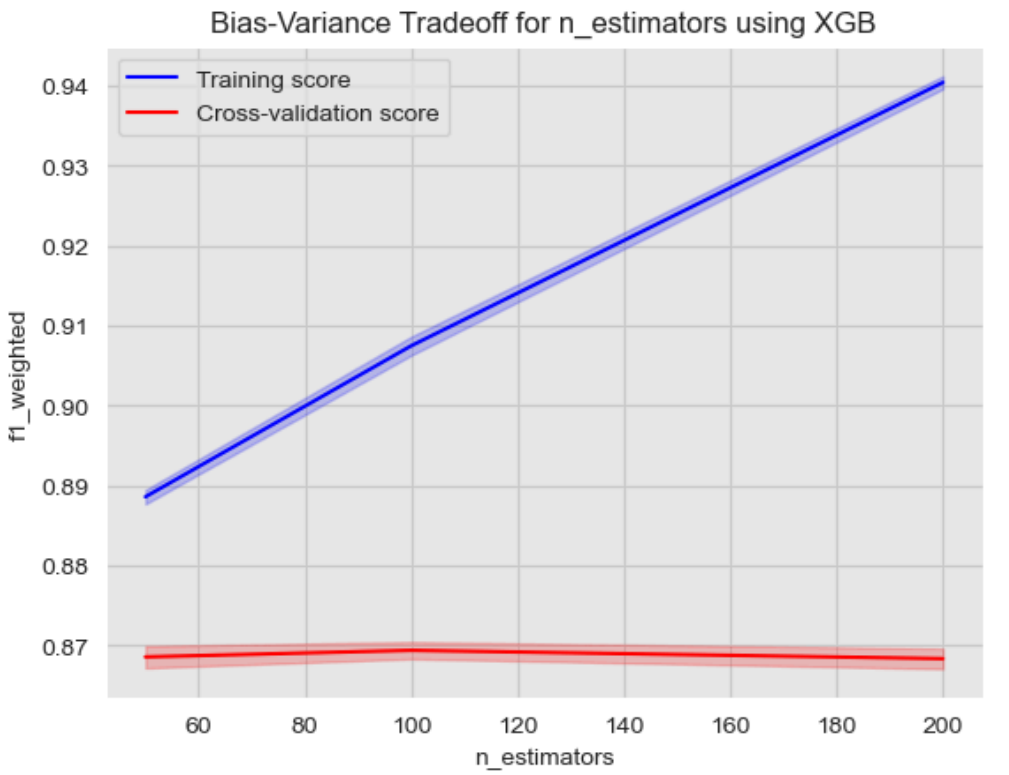
	precision	recall	f1-score	support
0	0.91	0.99	0.95	89917
1	0.57	0.11	0.19	10083
accuracy			0.90	100000
macro avg	0.74	0.55	0.57	100000
weighted avg	0.88	0.90	0.87	100000

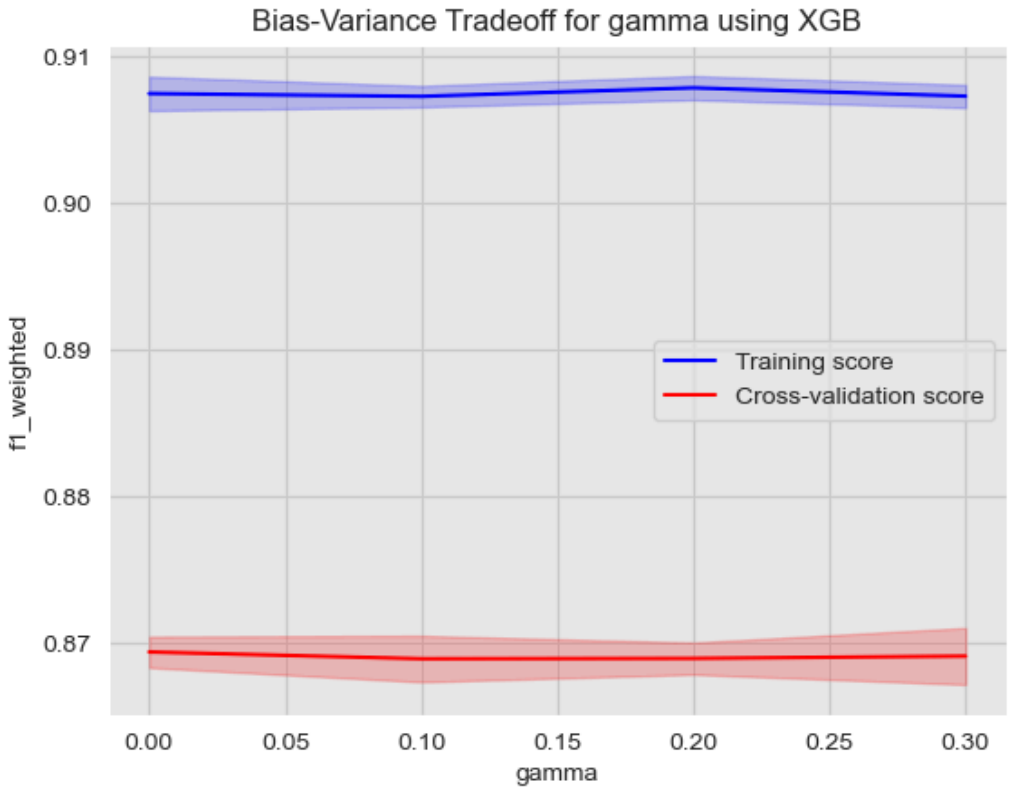
Train-Validation Curve

Here are some Train-Validation Curves that we further used for the hyperparameter tuning process:






Bias-Variance Tradeoff for n_estimators using XGB
Training score
Cross-validation score
f1_weighted
0.94
0.93
0.92
0.91
0.90
0.89
0.88
0.87
60
80
100
120
140
160
180
200
n_estimators

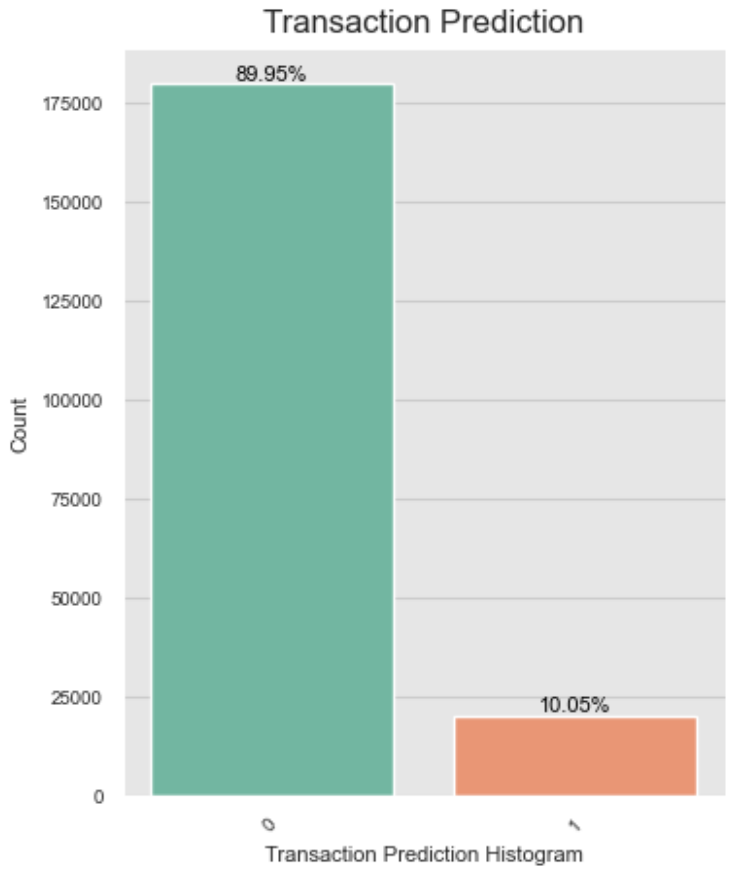

Bias-Variance Tradeoff for gamma using XGB
Training score
Cross-validation score
f1_weighted
0.91
0.90
0.89
0.88
0.87
0.00
0.05
0.10
0.15
0.20
0.25
0.30
gamma

Bias-variance Analysis

- mean square error: 0.09885829999999998
- bias: 0.092098
- var: 0.0067603000000000055
- Estimated Eout: 0.09885830000000001

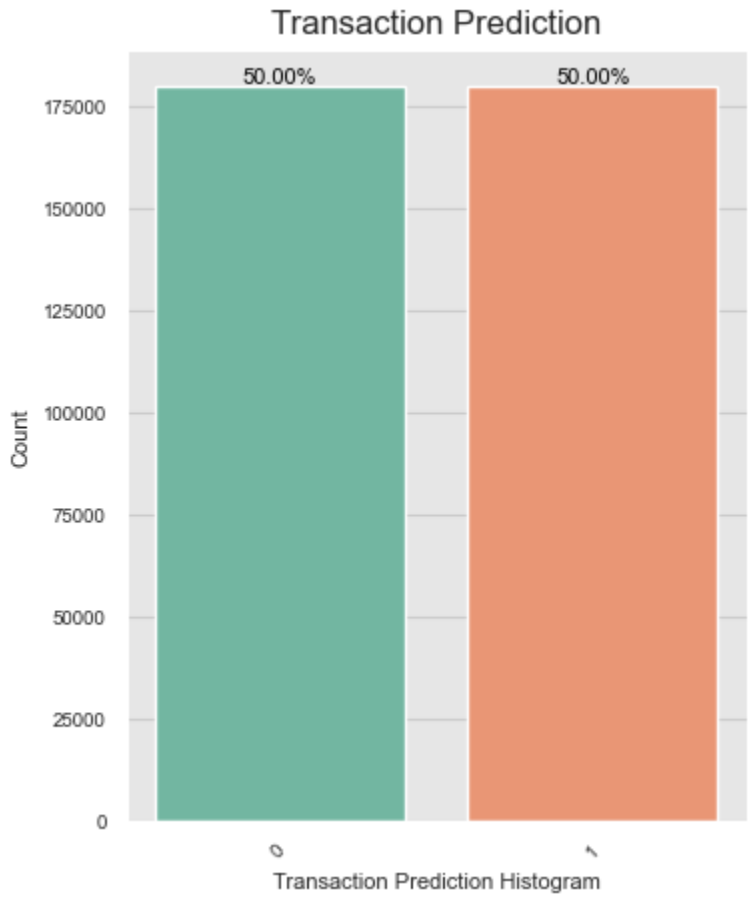
The problem of big difference between accuracy and weighted F1 score

It's noticed there is a big difference between accuracy and weighted F1-score that happened due to the problem of unbalanced classes in the dataset because the number of samples in class zero is way more than number of samples in class one, as you can see in the following chart:



So, one of the solutions to this problem is oversampling which is increasing the number of instances in the minority class. Techniques like SMOTE

(Synthetic Minority Over-sampling Technique) generate synthetic samples rather than simply duplicating existing ones. After applying SMOT, the size of the dataset increased to 359804 and the classes became balanced, as you can see in the following chart:



but unfortunately, the problem of small micro average still exists and also another problem appeared which is getting a high bias value and variance equal to zero which means the model is so simple to solve the problem so I tried to use a higher order model (with more features => complex target features) but that didn't solve the problem and the results are the following:

```
Classification Report of LogisticRegression(C=1, max_iter=10000, random_state=42, solver='saga') is:
              precision    recall  f1-score   support

           0       0.90      0.50      0.64     89951
           1       0.10      0.51      0.17     10049

    accuracy                           0.50    100000
   macro avg       0.50      0.50      0.41    100000
weighted avg       0.82      0.50      0.60    100000
```

Still, the difference between the weighted micro F1-score and accuracy is big as it's now 0.1 and previously it was 0.05.

another solution we tried was undersampling (Reducing the number of instances in the majority class. However, this might lead to loss of information), after applying undersampling, the size of the dataset decreased to 40196 the classes become balanced, as you can see in the following chart:

It gives results almost like oversampling but with lower accuracy and lower weighted F1-score.

Let's try another way to solve this problem, set the weights for each class inversely proportional to the number of samples of each class.
class_weight = {0: 1 - percentage_of_zeros, 1: 1 - percentage_of_ones}
The results are:

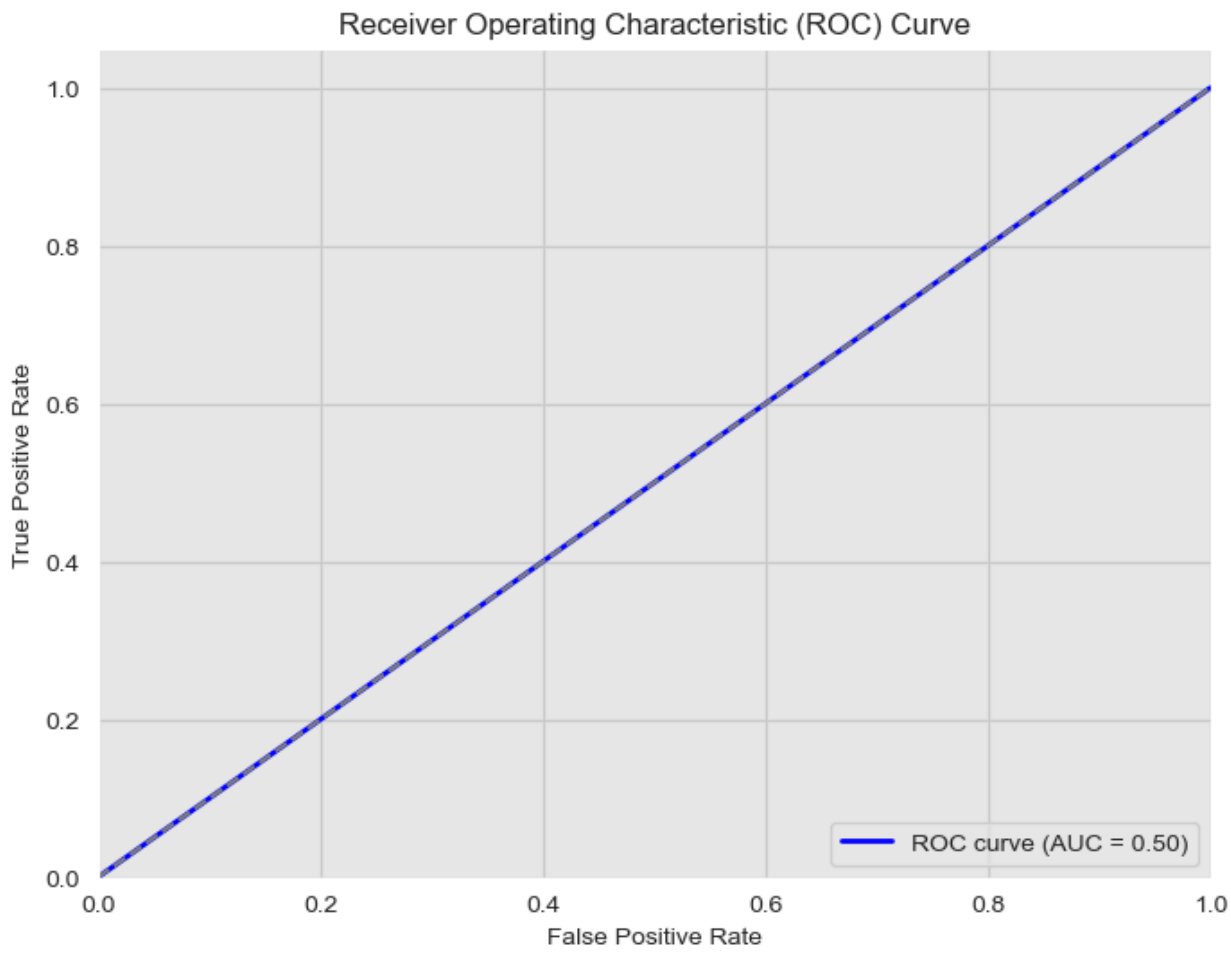
```
Classification Report of LogisticRegression(C=0.5, class_weight={0: 0.10048749999999995, 1: 0.8995125},
                   max_iter=10000, random_state=42, solver='saga') is:
              precision    recall  f1-score   support

           0       0.90      0.50      0.65     35980
           1       0.10      0.49      0.17      4020

    accuracy                           0.50     40000
   macro avg       0.50      0.50      0.41     40000
weighted avg       0.82      0.50      0.60     40000
```

Also, we tried class_weight = {0: 1/percentage_of_zeros, 1: 1/percentage_of_ones}
and we got the following results:

```
Classification Report of LogisticRegression(C=0.1,
                    class_weight={0: 1.1117163789174105, 1: 9.951238929246692},
                    max_iter=10000, random_state=42) is:
                precision    recall  f1-score   support

             0       0.90      0.51      0.65     89951
             1       0.10      0.50      0.17     10049

      accuracy                           0.51    100000
     macro avg       0.50      0.50      0.41    100000
  weighted avg       0.82      0.51      0.60    100000

  F1 Score of LogisticRegression(C=0.1,
                    class_weight={0: 1.1117163789174105, 1: 9.951238929246692},
                    max_iter=10000, random_state=42) is: 0.6011844568047079
```

but it's noticed that in case we set the class weights we can now classify some points of class one correctly instead of classify all of them wrong but still the difference between the accuracy and F1-score is big which is 0.09.

Using ROC-AUC matric without sampling and without setting the class weights. We got almost the same results for all models: AUC=0.5



even though the accuracy is almost 90% for all models the area under the ROC curve is 0.5 which is a small area that means the rate of the true positive rate equals the false positive rate and that happened because all our models classify the points of class one as class zero (FPR) with the same rate of classifying the points of class zeros as class zero (TPR).

Conclusion

comparison of evaluation metrics and bias-variance then choose the best model

Model Name	MSE	Bias	Variance	Estimated Eout
Logistic Regression	0.10049	0.10049	0	0.10049
SVM	0.1202861	0.10426	0.016026	0.120286

Random Forest	0.10085995	0.0893112	0.01154867	0.1008599
Adaboost	0.10049	0.10049	0	0.10049
XGB	0.09885829	0.092098	0.0067603	0.0988583

Model Name	Accuracy	Macro Avg F1-score	Weighted avg F1-score
Base model	0.90	0.47	0.85
Logistic Regression	0.90	0.47	0.85
SVM	0.88	0.49	0.85
Random Forest	0.90	0.57	0.87
Adaboost	0.90	0.48	0.85
XGB	0.90	0.57	0.87