Assessment Task 3: Data mining in action

Student Name: Khoi Huynh

Student Number: 24902037

Table of Contents

Contents

1. Attribute Description

Attribute Name	Description
age	Age of person
job	Type of job
marital	Marital status
education	Education level
default	Credit in default
housing	Housing loan
loan	Personal loan
contact	Contact communication type
month	Last contact month of year
day_of_week	Last contact day of the week
duration	Last contact duration
campaign	Number of contacts performed during the campaign
passed days	Number of days that passed by after the client was last contacted from a previous campaign
previous	Number of contacts performed before the campaign
poutcome	Outcome of the previous marketing campaign
variation rate	Employment variation rate – quarterly indicator
price index	Consumer price index – monthly indicator
confidence index	Consumer confidence index – monthly indicator
euribor3m	Euribor 3-month rate – daily indicator
no.employed	Number of employees – quarterly indicator
subscribed	Subscribed a term deposit
state	Name of the state

2. Data mining Problem

The business problem of the given data mining task is to predict whether or not the client will subscribe to a term deposit. The given dataset consists of various data types including continuous and discrete numerical features as well as categorical features. The table above provides a more detailed description of each of the input features. The output of this data mining task will be the “subscribed” binary feature where 1 indicates that the client did subscribe to a term deposit and the opposite is implied for 0.

To achieve the data mining task, various classifiers will be built and compared using performance metrics such as the confusion table, F1 score, and ROC-AUC curve. The following classifiers will be built in the later section of this report:

- Decision Tree
- Random Forest
- Support Vector Machine
- K-Nearest Neighbor
- Neural Network
- Logistic Regression
- Gradient Boosting

3. Data Pre-processing and Data Exploration

Before describing the data pre-processing and data exploration steps, the following libraries and dependencies are required:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn

from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.preprocessing import OrdinalEncoder, LabelEncoder, StandardScaler, MinMaxScaler
from sklearn.impute import SimpleImputer
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, HistGradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score, confusion_matrix, ConfusionMatrixDisplay, RocCurveDisplay, classification_report, roc_curve, auc
from sklearn.utils import compute_sample_weight, compute_class_weight

from imblearn.over_sampling import SMOTE

import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
```

[458] ✓ 0.0s Python

The usage of each of the libraries will be more evident in the **'Model Building'** section. First, data must be retrieved from the corresponding dataset as shown in Figure 1 below.

Data Collection

```
test_dataset = pd.read_csv(r"C:\Users\X1 Carbon\Downloads\Assignment3-Unknown-Dataset.csv")
train_validate_dataset = pd.read_csv(r"C:\Users\X1 Carbon\Downloads\Assignment3-Marketing-Dataset.csv")
```

[558] ✓ 0.1s Python

```
train_validate_dataset.head(3)
```

[559] ✓ 0.0s Python

	row ID	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	emp.var.rate	cons.price.idx	cons
0	Row0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	261	1	999	0	nonexistent	1.1	93.994	
1	Row1	56	services	married	high.school	no	no	yes	telephone	may	mon	307	1	999	0	nonexistent	1.1	93.994	
2	Row2	45	services	married	basic.9y	unknown	no	no	telephone	may	mon	198	1	999	0	nonexistent	1.1	93.994	

Figure 1. Data Collection

Immediately, the data quality issue of missing values can be identified. Since missing values are encoded as 2 separate labes which are: '?' and 'unknown', this will be challenging to keep track of during data cleaning since each steps will have to be carried

out twice, one for handling the missing labels encoded as ‘unknown’ and an addition step for the missing labels ecoded as ‘?’. Furthermore, even though the missing labels are labelled specifically as ‘?’ and ‘unknown’, the pandas library does not recognize these labels as missing values natively. Instead, it treats them as regular strings, which is undesired. Therefore, we will be converting the labels ‘?’ and ‘unknown’ into 1 single unified missing label that the pandas library natively recognizes as missing values for easier management, as shown in Figure 2.

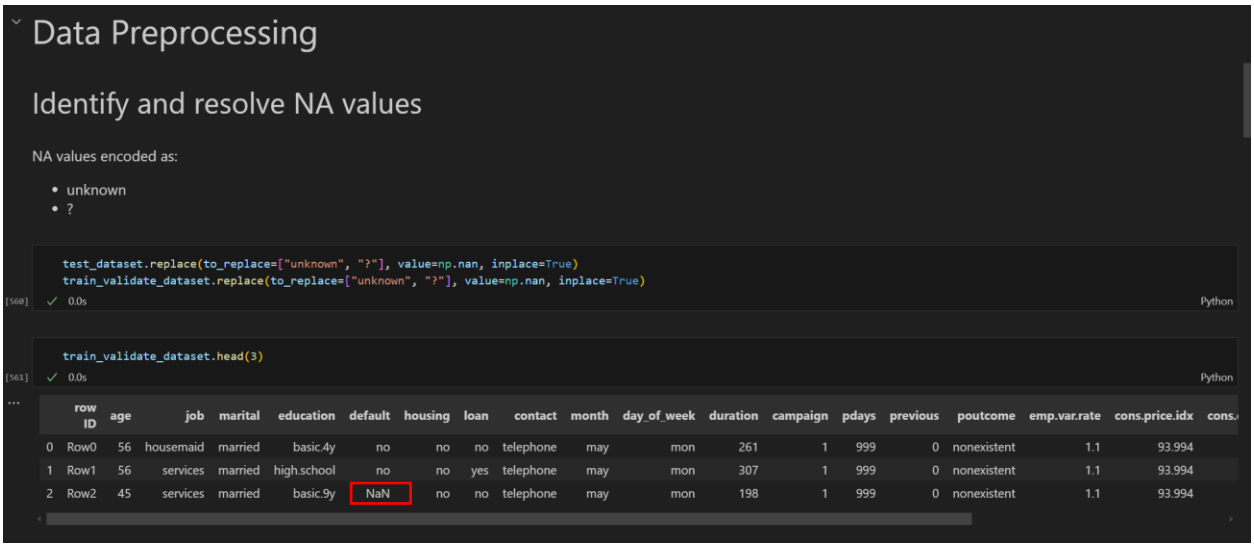


Figure 2. Missing values encoding

Depending on which feature we are dealing with, missing values will be resolved differently. Also, before starting to impute the missing values, we need to split the train_validate_dataset into 2 separate datasets where the ratio between the training and validation datasets is 70 : 30 as shown in Figure 3 below. The data sets are split using a normal partition since later on, during hyperparameter tuning, cross-validation is also incorporated into it. This is to avoid data leakage from the training dataset onto the validation dataset.

Figure 3. Split into train and validation datasets

The overview of the training dataset can be found in Figure 4a and Figure 4b. Some features may appear to have their datatype mismatched, for example, you would expect

the number of passed days to be an integer, but it turns out to be a string. This is an indicator that there are missing values within the feature since now missing values are encoded as ‘NaN’. For clearer and more explicit count of missing values, refer to Figure 4b.

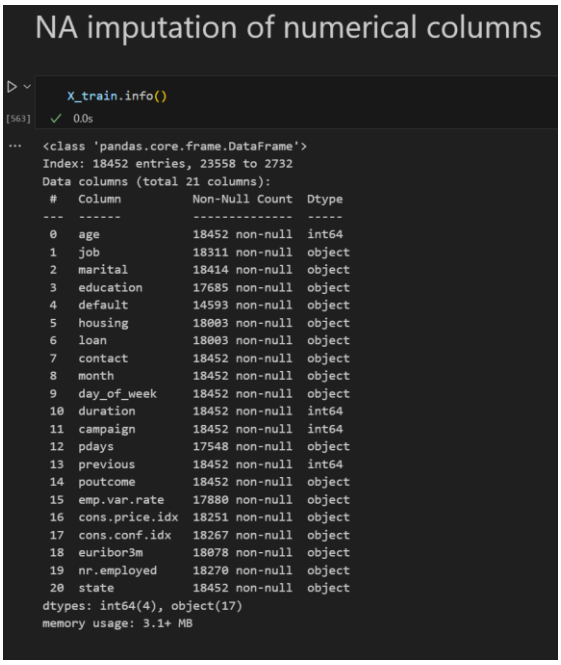


Figure 4a. Overview of training dataset

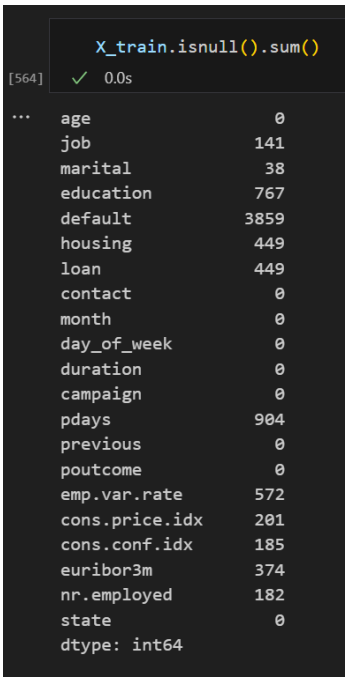


Figure 4b. Missing values count

We will start imputing missing values in numerical features such as ‘nr.employed’, ‘euribor3m’, etc. Since the datatype of these features is currently string, we will need to convert them to float before imputing.

```
X_train = X_train.astype({'nr.employed' : float,
                          'euribor3m' : float,
                          'cons.conf.idx' : float,
                          'cons.price.idx' : float,
                          'emp.var.rate' : float,
                          'pdays' : float})

X_validate = X_validate.astype({'nr.employed' : float,
                                'euribor3m' : float,
                                'cons.conf.idx' : float,
                                'cons.price.idx' : float,
                                'emp.var.rate' : float,
                                'pdays' : float})

test_dataset = test_dataset.astype({'nr.employed' : float,
                                    'euribor3m' : float,
                                    'cons.conf.idx' : float,
                                    'cons.price.idx' : float,
                                    'emp.var.rate' : float,
                                    'pdays' : float})

X_train.dtypes
```

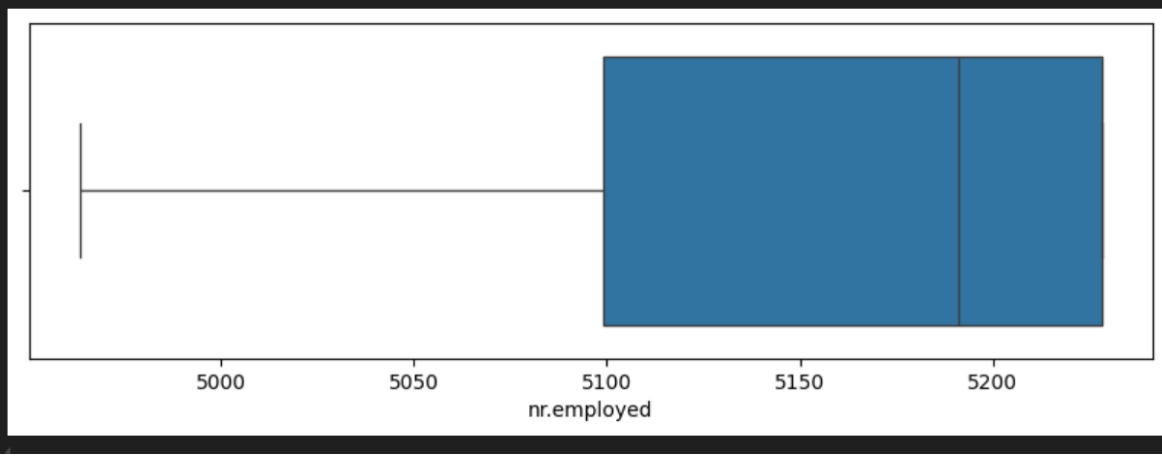
[565] ✓ 0.0s

The distributions of each of the numeric features mentioned above can be seen in the following figures below:

```
plt.figure(figsize=(10,3))
sns.boxplot(data=X_train, x="nr.employed")
```

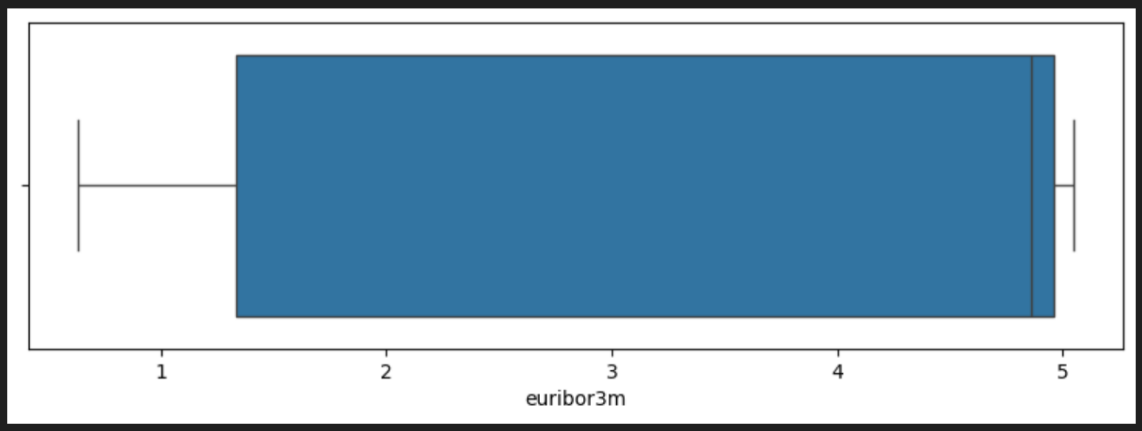
[566] ✓ 0.0s

```
<Axes: xlabel='nr.employed'>
```



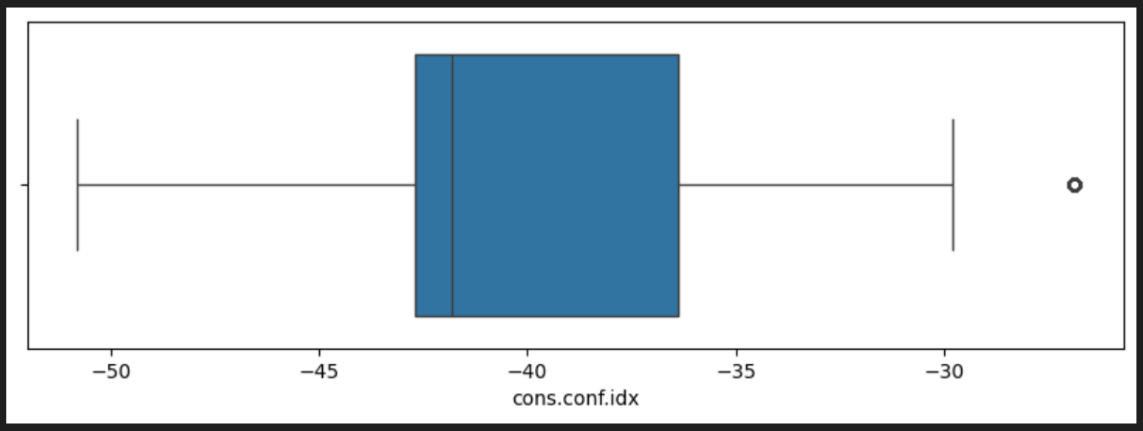
```
[567]  plt.figure(figsize=(10,3))
       sns.boxplot(data=X_train, x="euribor3m")
       ✓ 0.0s
```

```
<Axes: xlabel='euribor3m'>
```




```
[568]  plt.figure(figsize=(10,3))
       sns.boxplot(data=X_train, x="cons.conf.idx")
       ✓ 0.0s
```

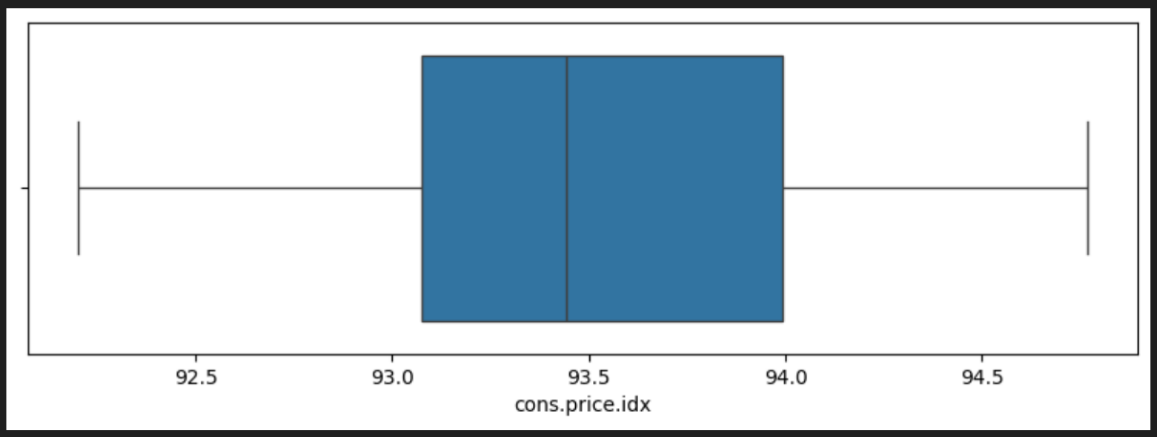
```
<Axes: xlabel='cons.conf.idx'>
```



```
plt.figure(figsize=(10,3))
sns.boxplot(data=X_train, x="cons.price.idx")
```

[569] ✓ 0.1s

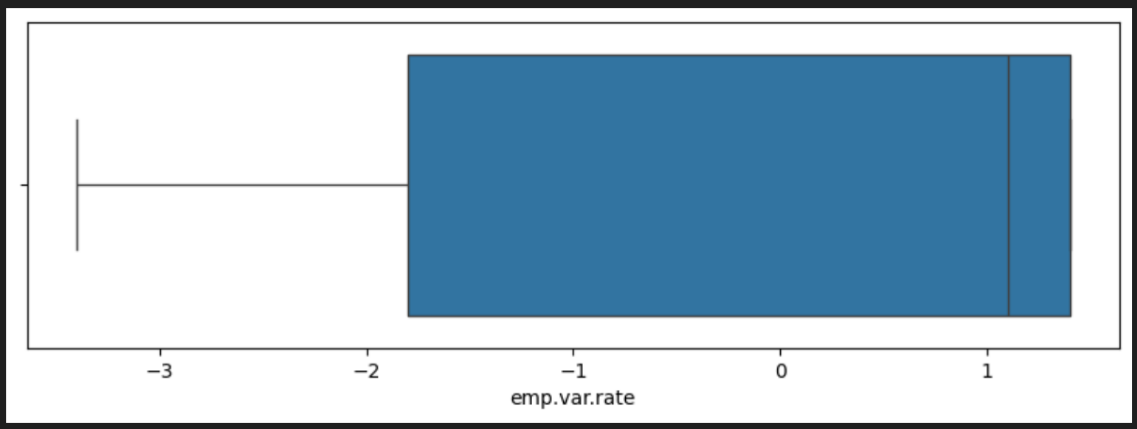
```
<Axes: xlabel='cons.price.idx'>
```




```
plt.figure(figsize=(10,3))
sns.boxplot(data=X_train, x="emp.var.rate")
```

[570] ✓ 0.1s

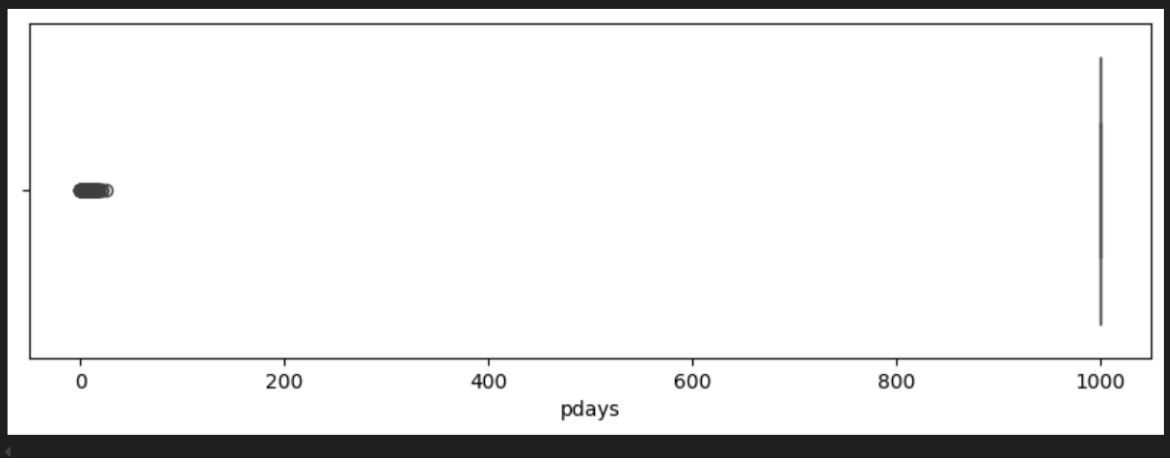
```
<Axes: xlabel='emp.var.rate'>
```



```
plt.figure(figsize=(10,3))
sns.boxplot(data=X_train, x="pdays")
```

[571] ✓ 0.1s

```
<Axes: xlabel='pdays'>
```

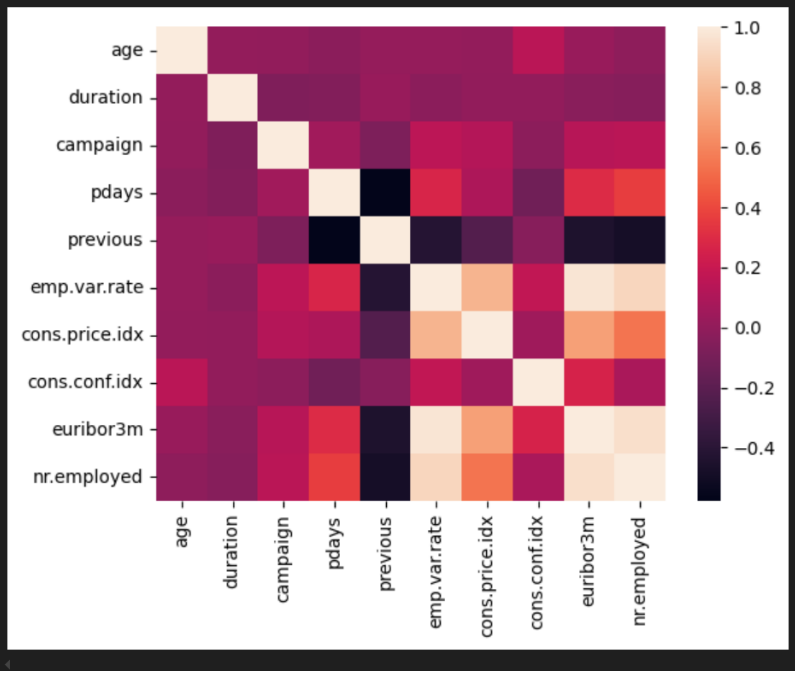


Pearson Correlation

```
temp = X_train.select_dtypes(include=['int64', 'float64'])
sns.heatmap(temp.corr(method='pearson'))
```

[583] ✓ 0.2s

```
<Axes: >
```



Starting with the ‘pdays’ feature, as seen from its distribution above, an overwhelming proportion of the values have the value 999. Additionally, some outliers can be seen based on the distribution. As a result, I have decided to impute the missing values with the mode since the number of outliers is minuscule, along with more than 95% of the values being equal to 999, as shown in Figure 5.

```
[30]  ✓ 0.0s

pdays_imputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')

# Impute NA values in pdays column
X_train['pdays'] = pdays_imputer.fit_transform(X_train[['pdays']])
X_validate['pdays'] = pdays_imputer.transform(X_validate[['pdays']])
test_dataset['pdays'] = pdays_imputer.transform(test_dataset[['pdays']])
```

Figure 5. Impute NaN values with mode in ‘pdays’ column

For the remaining numeric columns, such as ‘euribor3m’ or ‘cons.price.idx’, a DecisionTreeRegressor will be built to impute missing values for each of the features. In order to gauge the goodness of fit of these models, performance metrics such as R^2 and MSE will be employed. R^2 has a range from 0 to 1, where 1 indicates a perfect model and 0 indicates that the model is no better than just predicting the mean.

```
[31]  ✓ 0.0s

# Train model to impute NA values in emp.var.rate
X_numeric = X_train.select_dtypes(include=['float64', 'int64'])
X_numeric = X_numeric.dropna()
validate_numeric = X_validate.select_dtypes(include=['float64', 'int64'])
validate_numeric = validate_numeric.dropna()

response = X_numeric['emp.var.rate']
predictor = X_numeric.drop(columns=['emp.var.rate'])

emp_var_rate_imputer = DecisionTreeRegressor(criterion='squared_error', random_state=0)
emp_var_rate_imputer.fit(predictor, response)

print(f"MSE: {mean_squared_error(response, emp_var_rate_imputer.predict(predictor))}") # In-sample Performance
print(f"R2: {r2_score(response, emp_var_rate_imputer.predict(predictor))}")

response = validate_numeric['emp.var.rate']
predictor = validate_numeric.drop(columns=['emp.var.rate'])

print(f"MSE: {mean_squared_error(response, emp_var_rate_imputer.predict(predictor))}") # Out-sample Performance
print(f"R2: {r2_score(response, emp_var_rate_imputer.predict(predictor))}")

...  MSE: 1.8701060409981618e-26
     R2: 1.0
     MSE: 1.8887146946235862e-26
     R2: 1.0
```

Figure 6. DecisionTreeRegressor to impute NaN values in ‘emp.var.rate’ column

At this point, all the missing values in the numerical features have been resolved. However, since we are going to be building distance-based models such as KNN or SVM, their predictive performance is going to be hindered due to the features having different scales. For example, as seen in the distribution of each feature above, ‘emp.var.rate’ is on the single-digit scale, whereas ‘nr.employed’ is on the thousand scale. Due to how these models work where the classification will be dictated by either the distance from the data point to its neighbor or the separating hyperplane / line. By introducing features of different scales together, the models will unintentionally be biased, where features on a larger scale may be interpreted as more important.

To correct this issue, we will have to normalize / standardize the data. In Figure 7 below, we normalize the data since KNN and SVM models do not require the data to assume a normal distribution. However, for logistic regression models, standardization is preferred. Tree-based models like Decision Tree or Random Forest, on the other hand, do not require the data to be on the same scale at all since they operate purely on information gain and entropy. Before normalizing, we must convert the data types of the numerical features from int to float to avoid accidentally truncating the decimal portion.

Figure 7. Normalization

Next, we will be imputing missing values in the categorical features. Since the models in the sklearn library only accept numerical values, we will first convert ordinal categorical features like ‘education’ or ‘month’ into numeric values and perform one-hot encoding to ‘nominal’ categorical features like ‘job’ and ‘state’, as shown in the three figures below.

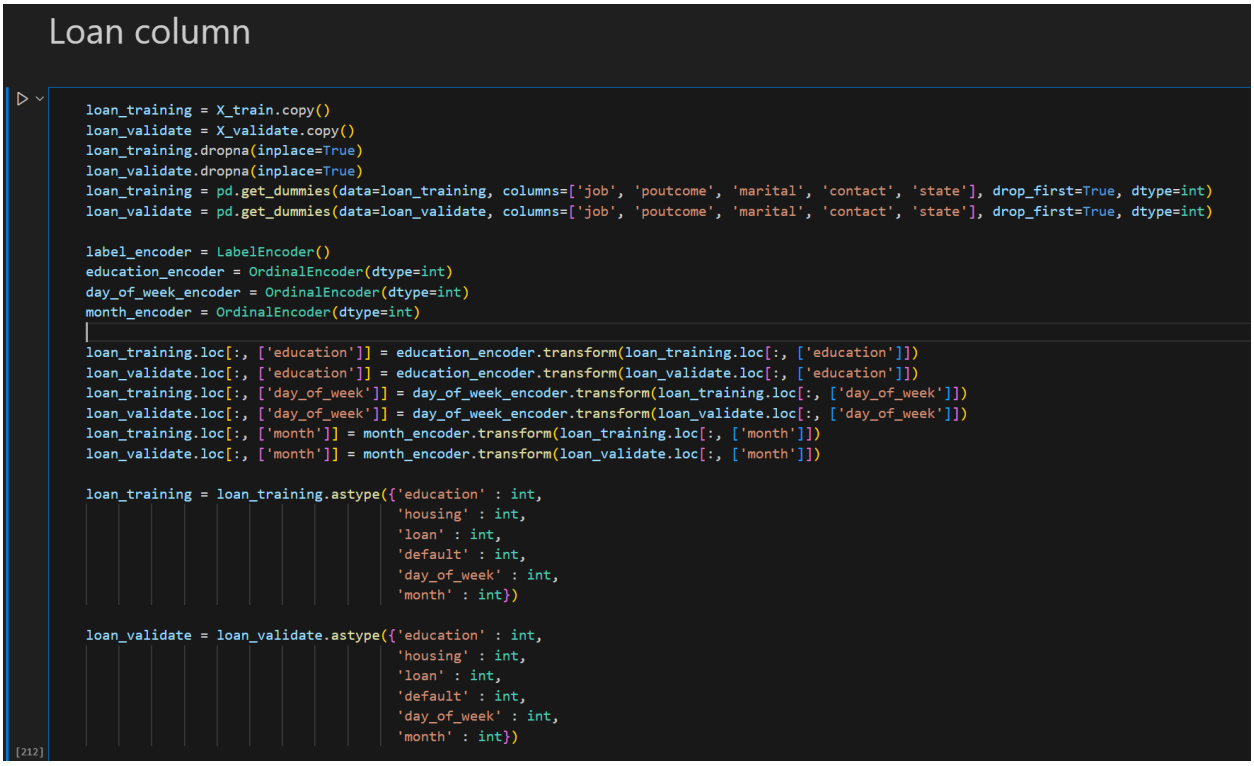


Figure 8a. Perform one-hot encoding and label encoding

	age	education	default	housing	loan	month	day_of_week
17451	0.259740	6	0	1	1	7	0
20212	0.155844	5	0	1	0	6	2
24505	0.142857	3	0	0	0	8	4
6804	0.155844	6	0	0	1	4	3
23543	0.311688	6	0	1	0	4	1
17755	0.220779	5	0	0	0	5	0
7199	0.350649	5	0	1	0	4	0
6304	0.142857	6	0	0	0	4	4
2476	0.272727	2	0	1	1	6	1
23901	0.883117	5	0	0	0	1	3
25646	0.181818	6	0	1	0	4	4
4390	0.207792	5	0	1	1	6	2
4831	0.181818	6	0	0	0	6	0
15908	0.298701	3	0	1	0	7	3
22747	0.389610	5	0	1	0	6	1

Figure 8b. Result data frame (1)

job_blue-collar	job_entrepreneur	job_housemaid	job_management	job_retired	job_self-employed	job_services	job_student	job_technician	job_unemployed	marital_married	marital_single
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0	0	1

Figure 8c. Result data frame (2)

Notice that when performing one-hot encoding, I have set the parameter ‘drop_first’ to be True. The reason is to avoid the problem of multicollinearity when I try to build a logistic regression model later. For other models, I set the parameter ‘drop_first’ back to False. Before building the models to impute missing values in the feature ‘loan’, I noticed that there is a significant unbalance in the distribution of the ‘loan’ feature as seen below.

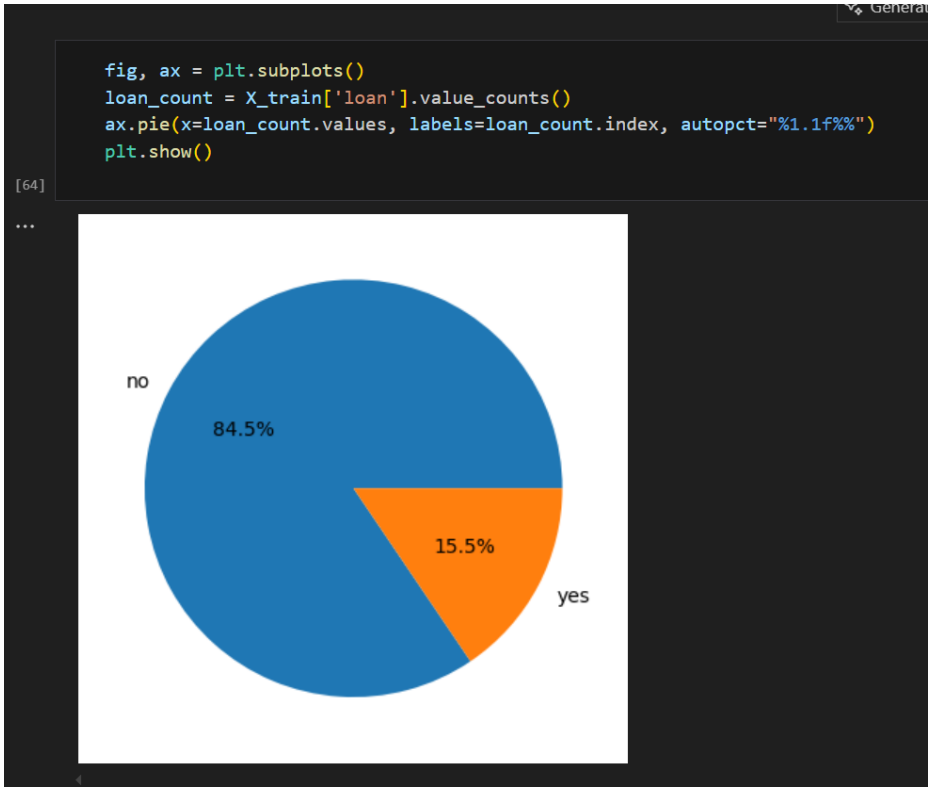


Figure 9. Distribution of feature ‘loan’

To address this class imbalance, I performed the SMOTE technique to even out the distribution. In case SMOTE is not effective, I also tried to compute and assign weights to each class label. The higher the weights, the more importance the model will give to the corresponding class label.

```
response = loan_training['loan']
predictor = loan_training.drop(columns=['loan']) #
w_train = compute_sample_weight(class_weight='balanced', y=response)
class_weight_dict = dict(zip(response.unique(), w_train))
smote = SMOTE(random_state=1)
new_predictor, new_response = smote.fit_resample(predictor, response)
response_valid = loan_validate['loan']
predictor_valid = loan_validate.drop(columns=['loan']) #
```

Figure 10. SMOTE and class weights

After dealing with the class imbalance, we can finally start building different models to impute the missing values in the ‘loan’ column. However, despite many attempts, I was not able to train the model up to a satisfactory degree of accuracy, as the issue of overfitting always persisted. While the model can predict the class ‘0’ relatively with ease, it struggles to properly identify the class ‘1’. I tried many approaches to fix the overfitting problem, from performing cross-validation to hyperparameter tuning, but none turned out to be effective. This may be because of some potential oversight.

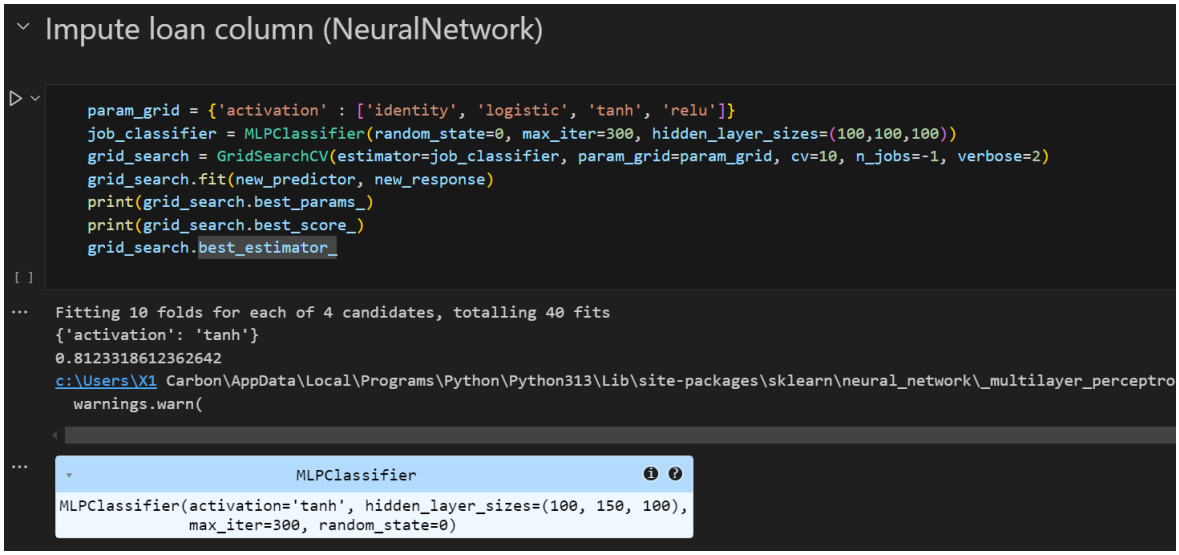


Figure 11. ‘Best’ model for imputing missing values in ‘loan’ feature

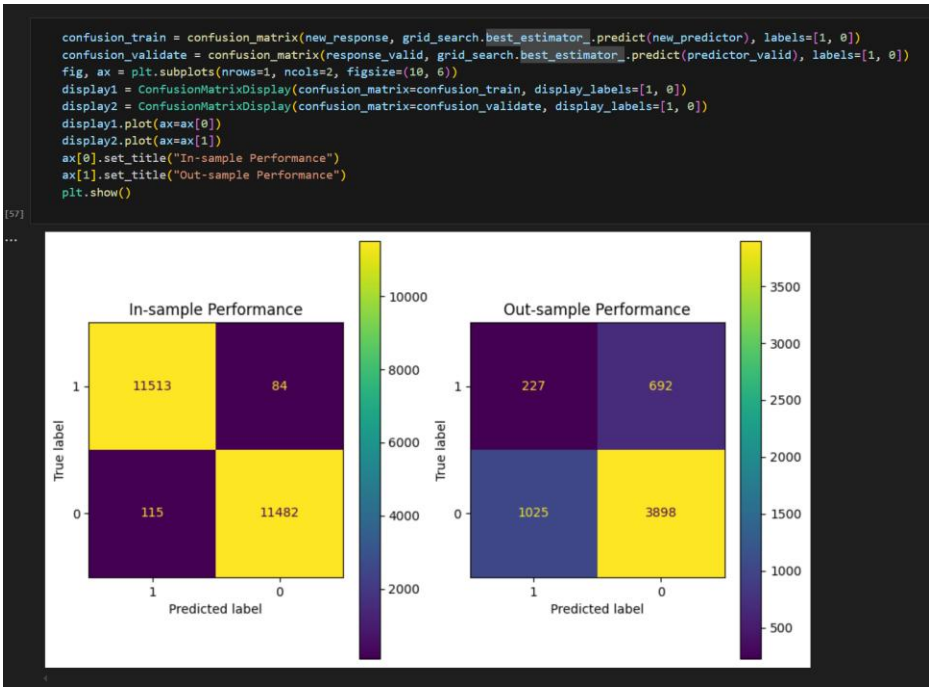


Figure 12. Confusion matrix for ‘loan’ feature

This issue is not limited only to the ‘loan’ feature, but for the majority of the categorical features. Finally, I decided to drop all the missing values in all categorical features. The justification for this is that if we keep using inadequate models to impute the missing values, it would only worsen the situation by introducing more noise. I have considered replacing the missing values with the mode; however, when the proportion of missing values is considerable, simply replacing them with the mode will not effectively capture the relationship and will only introduce bias into the model, potentially misguiding the model from the original relationship.

4. Feature Selection

For the models to be trained and perform effectively, feature selection is performed to only include features that are useful in separating the classes. Introducing irrelevant features into the models will not only significantly degrade the performance of the said model. The chosen feature selection method is the backward selection method, implemented via the SequentialFeatureSelector class in the mlxtend library.

Backward selection is an iterative process. It will start with a full model, a model trained on every feature, and repeatedly remove the least significant feature until the desired number of features is reached. This can be seen in the figure below. It is important to note that the **selected features will be different depending on what estimator / model is used**.

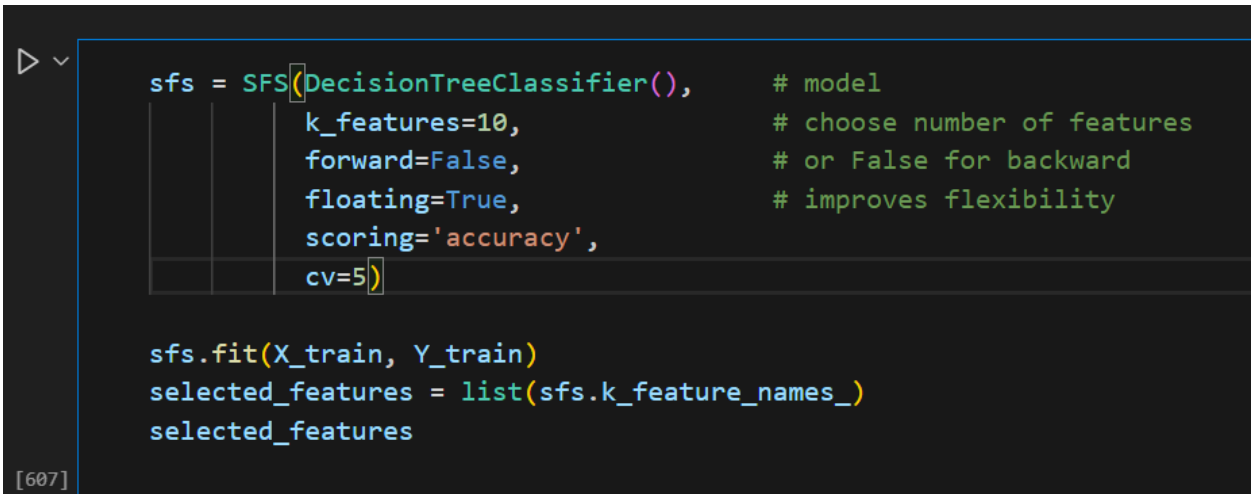


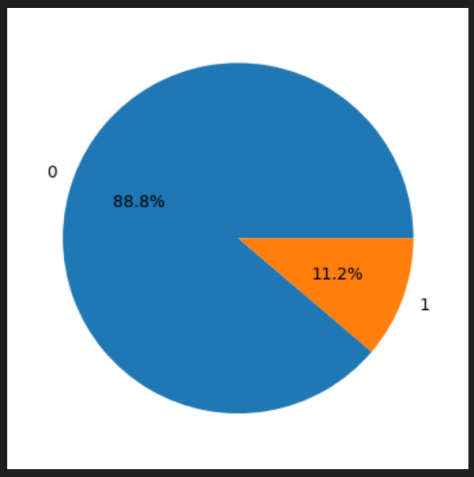
Figure 13. Feature selection (backward selection)

5. Model Building

The models will be trained on the training dataset, and their performance will be gauged using the validation dataset. The hyperparameters of the models will be tuned using the GridSearchCV class, which also incorporates cross-validation into the tuning process in the sklearn library. Additionally, since there is a class imbalance in the ‘subscribed’ column, I will use SMOTE to generate synthetic records of the minority class to even out the distribution.

```
fig, ax = plt.subplots()
subscribed_count = Y_train.value_counts()
ax.pie(x=subscribed_count.values, labels=subscribed_count.index, autopct="%1.1f%%")
plt.show()
```

[183] ✓ 0.0s



- **Decision Tree:**

```
w_train = compute_sample_weight(class_weight='balanced', y=Y_train)
class_weight_dict = dict(zip(response.unique(), w_train))

param_grid = {'criterion' : ['gini', 'entropy', 'log_loss'],
              'min_samples_leaf' : [2, 3, 4, 5], #1
              'max_depth' : [15, 10]} #20

model = DecisionTreeClassifier(random_state=0, class_weight='balanced')
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(X_train[selected_features], Y_train)
print(grid_search.best_params_)
print(grid_search.best_score_)
grid_search.best_estimator_
```

[46] ✓ 4.8s

```
Fitting 10 folds for each of 24 candidates, totalling 240 fits
{'criterion': 'gini', 'max_depth': 15, 'min_samples_leaf': 2}
0.870468789385334
```

DecisionTreeClassifier

```
DecisionTreeClassifier(class_weight='balanced', max_depth=15,
                       min_samples_leaf=2, random_state=0)
```

Since tree-based models classify purely based on which feature increases the information gain the most, we are interested in seeing the importance of the features, which can be shown below.

```
importances = grid_search.best_estimator_.feature_importances_
importance_df = pd.DataFrame({'feature': selected_features, 'importance': importances})
importance_df = importance_df.sort_values(by='importance', ascending=False)
importance_df
```

[603]

	feature	importance
5	nr.employed	0.681698
4	cons.conf.idx	0.195517
2	emp.var.rate	0.040529
9	poutcome_success	0.038697
1	month	0.014434
6	job_entrepreneur	0.013261
7	poutcome_failure	0.009129
3	cons.price.idx	0.004083
8	poutcome_nonexistent	0.002652
0	default	0.000000

Figure 14. Feature importances of decision tree model

```
confusion_train = confusion_matrix(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), labels=[1,0])
confusion_validate = confusion_matrix(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), labels=[1, 0])
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 3))
display1 = ConfusionMatrixDisplay(confusion_matrix=confusion_train, display_labels=[1, 0])
display2 = ConfusionMatrixDisplay(confusion_matrix=confusion_validate, display_labels=[1, 0])
display1.plot(ax=ax[0])
display2.plot(ax=ax[1])
ax[0].set_title("In-sample Performance")
ax[1].set_title("Out-sample Performance")
plt.show()
```

[61] ✓ 0.2s

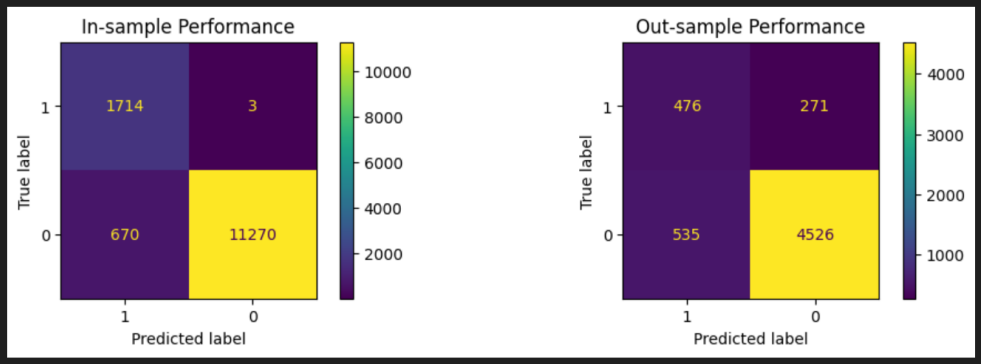


Figure 15. Confusion matrix of decision tree model

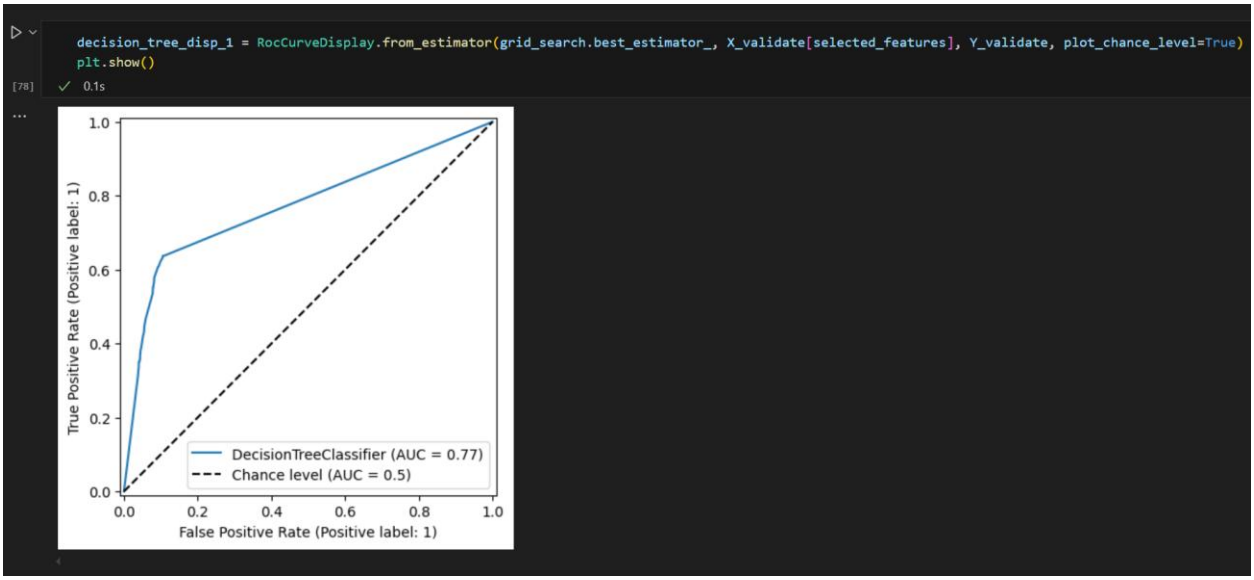


Figure 16a. ROC and AUC of decision tree model (validation)

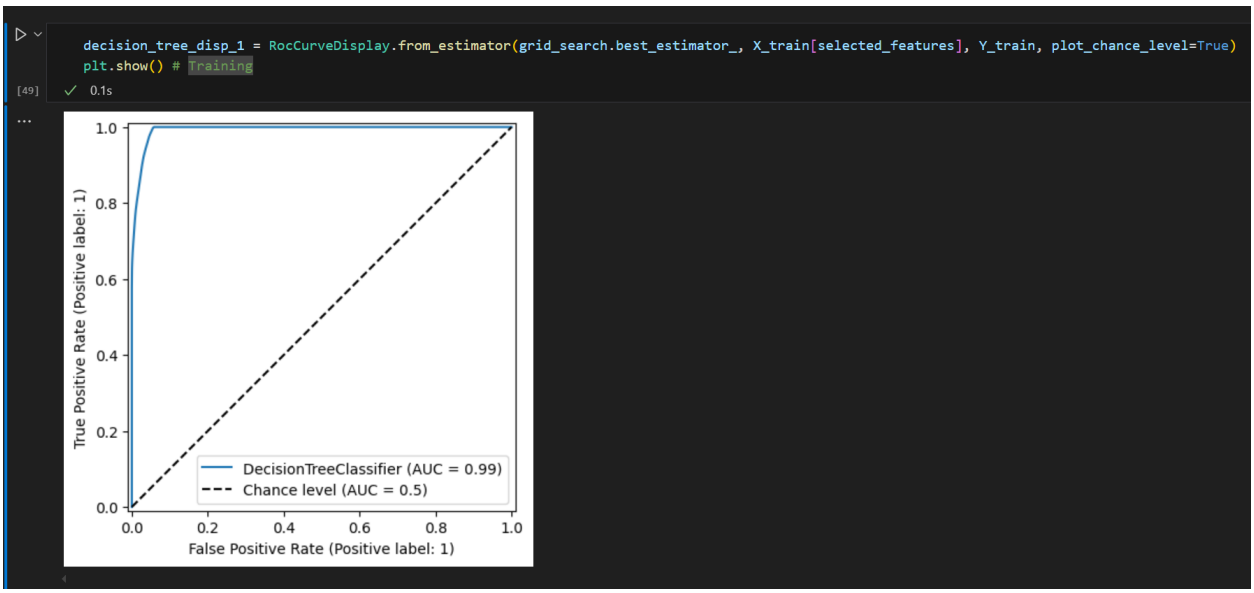


Figure 16b. ROC and AUC of decision tree model (training)

```
report = classification_report(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), output_dict=True)
pd.DataFrame(report)
```

[87] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.943506	0.470821	0.861226	0.707164	0.882712
recall	0.894290	0.637216	0.861226	0.765753	0.861226
f1-score	0.918239	0.541524	0.861226	0.729882	0.869788
support	5061.000000	747.000000	0.861226	5808.000000	5808.000000

Figure 17a. Classification report of decision tree model (validation)

```
report = classification_report(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), output_dict=True)
pd.DataFrame(report)
```

[201] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.999734	0.718960	0.950721	0.859347	0.964434
recall	0.943886	0.998253	0.950721	0.971069	0.950721
f1-score	0.971008	0.835894	0.950721	0.903451	0.954021
support	11940.000000	1717.000000	0.950721	13657.000000	13657.000000

Figure 17b. Classification report of decision tree model (training)

The Decision Tree model above is pruned by setting the parameter $\min\_samples\_leaf = 2$ to increase the minimum number of samples in each node before it can be further split to avoid overfitting. Additionally, I also set the parameter $\max\_depth = 15$ to restrict how deep the Decision Tree model can expand to avoid overfitting. Finally, the chosen criterion for how the Decision Tree model decides to split the nodes is based on the Gini index. This model has a Kaggle score of 0.57950.

- **Random Forest:**

```
w_train = compute_sample_weight(class_weight='balanced', y=Y_train)
class_weight_dict = dict(zip(response.unique(), w_train))

param_grid = {'criterion' : ['gini', 'entropy', 'log_loss'],
              'min_samples_leaf' : [1, 2, 3, 4, 5],
              'max_depth' : [20, 15, 10]}
model = RandomForestClassifier(random_state=0, bootstrap=True, n_estimators=100)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(new_predictor[selected_features], new_response)
print(grid_search.best_params_)
print(grid_search.best_score_)
grid_search.best_estimator_
```

[86] ✓ 6m 42.1s

```
Fitting 10 folds for each of 45 candidates, totalling 450 fits
{'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 1}
0.9374790619765495
```

RandomForestClassifier

```
RandomForestClassifier(criterion='entropy', max_depth=20, random_state=0)
```

Since Random Forest is also a tree-based model, we can give the importances of each feature below, to see how each of the features contributes to the splitting of nodes.

```
importances = grid_search.best_estimator_.feature_importances_
importance_df = pd.DataFrame({'feature': selected_features, 'importance': importances})
importance_df = importance_df.sort_values(by='importance', ascending=False)
importance_df
```

[89] ✓ 0.0s

	feature	importance
2	duration	0.394353
7	euribor3m	0.201977
6	emp.var.rate	0.116916
3	campaign	0.088160
0	age	0.073033
1	day_of_week	0.044170
4	pdays	0.033251
5	previous	0.026155
8	contact_cellular	0.014583
9	state_WA	0.007402

Figure 18. Feature importances of random forest model

```
confusion_train = confusion_matrix(new_response, grid_search.best_estimator_.predict(new_predictor[selected_features]), labels=[1,0])
confusion_validate = confusion_matrix(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), labels=[1, 0])
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 3))
display1 = ConfusionMatrixDisplay(confusion_matrix=confusion_train, display_labels=[1, 0])
display2 = ConfusionMatrixDisplay(confusion_matrix=confusion_validate, display_labels=[1, 0])
display1.plot(ax=ax[0])
display2.plot(ax=ax[1])
ax[0].set_title("In-sample Performance")
ax[1].set_title("Out-sample Performance")
plt.show()
```

[91] ✓ 0.6s

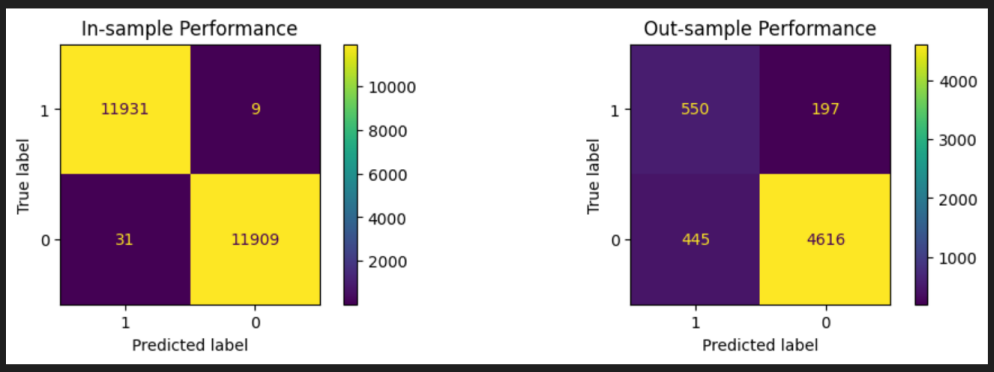


Figure 19. Confusion matrix of random forest model

```
random_forest_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_validate[selected_features], Y_validate, plot_chance_level=True)
plt.show()
```

[54] ✓ 0.2s

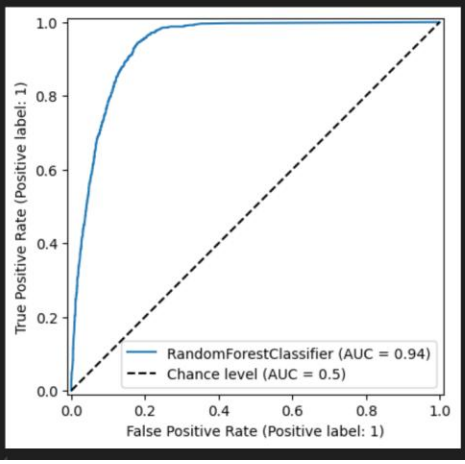


Figure 20a. ROC and AUC of random forest model (validation)

```
random_forest_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_train[selected_features], Y_train, plot_chance_level=True)
plt.show()
```

[55] ✓ 0.3s

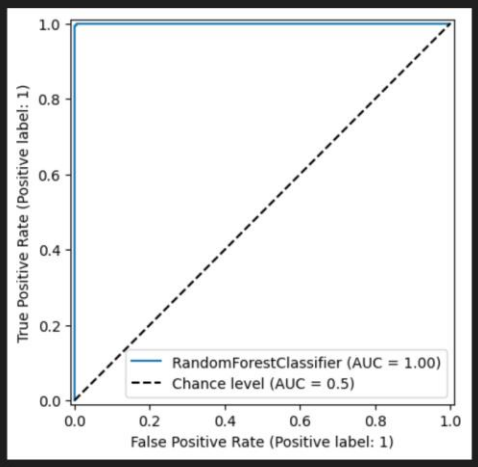


Figure 20b. ROC and AUC of random forest model (training)

```
report = classification_report(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), output_dict=True)
pd.DataFrame(report)
```

[93] ✓ 0.1s

	0	1	accuracy	macro avg	weighted avg
precision	0.959069	0.552764	0.889463	0.755917	0.906812
recall	0.912073	0.736278	0.889463	0.824176	0.889463
f1-score	0.934981	0.631458	0.889463	0.783219	0.895943
support	5061.000000	747.000000	0.889463	5808.000000	5808.000000

Figure 21a. Classification report of random forest model (validation)

```
report = classification_report(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), output_dict=True)
pd.DataFrame(report)
```

[204] ✓ 0.3s

	0	1	accuracy	macro avg	weighted avg
precision	0.999580	0.982215	0.997364	0.990897	0.997397
recall	0.997404	0.997088	0.997364	0.997246	0.997364
f1-score	0.998491	0.989595	0.997364	0.994043	0.997372
support	11940.000000	1717.000000	0.997364	13657.000000	13657.000000

Figure 21b. Classification report of random forest model (training)

Since Random Forest model is an ensemble model based on multiple independent trees, therefore the parameters are identical as the Decision Tree model. However, the values of the parameter are different. Noticeably, the Random Forest model now chooses to split the nodes based on the entropy level and information gain

compared to using the Gini index like the Decision Tree model. This model has a Kaggle score of 0.63088.

- **KNN:**

  When building a KNN model, the most important thing is to determine the most optimal k (number of neighbors) to use. After training the KNN model on various k and using cross validation to determine the mean score of each k, I have determined that $k = 7$ yields the lowest average error rate as seen in Figure 22 and Figure 23 below. The KNN model has a Kaggle score of 0.58058.

```
error_rate = []

for i in range(1, 40):
    knn = KNeighborsClassifier(n_neighbors=i)
    score = cross_val_score(knn, X_train[selected_features], Y_train, cv=10)
    error_rate.append(1-score.mean())
```

[113] ✓ 54.3s

```
plt.figure(figsize=(10,6))
plt.plot(range(1, 40), error_rate, color='blue', linestyle='dashed', marker='o', markerfacecolor='red', markersize=10)
plt.title('Error rate VS K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

[114] ✓ 0.1s

Figure 22. Determining the best k value

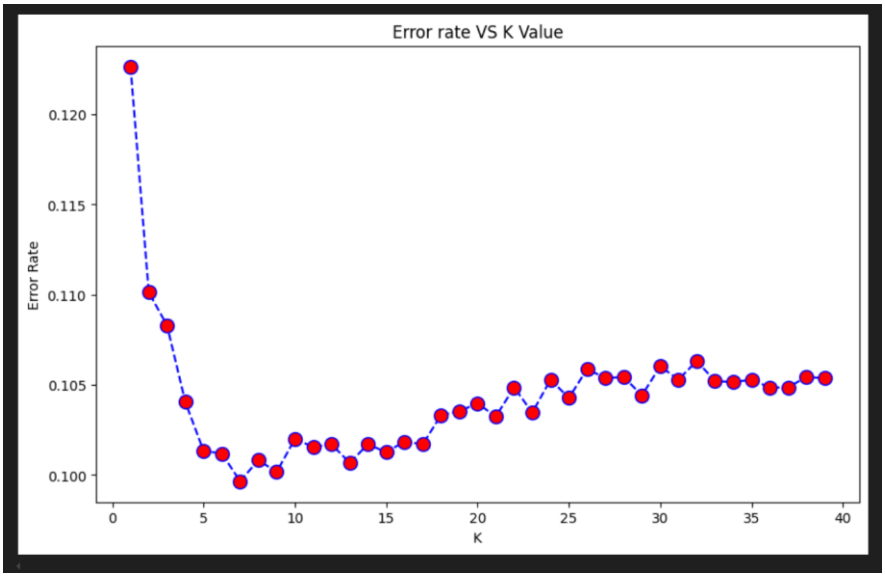


Figure 23. Average error rate of each k value

```
param_grid = {'n_neighbors' : [7],
              'weights' : ['distance', 'uniform']}

model = KNeighborsClassifier(n_jobs=-1)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(new_predictor[selected_features], new_response)
print(grid_search.best_params_)
print(grid_search.best_score_)
grid_search.best_estimator_
```

[115] ✓ 1.5s

```
Fitting 10 folds for each of 2 candidates, totalling 20 fits
{'n_neighbors': 7, 'weights': 'distance'}
0.8967336683417086
```

KNeighborsClassifier

```
KNeighborsClassifier(n_jobs=-1, n_neighbors=7, weights='distance')
```

```
confusion_train = confusion_matrix(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), labels=[1,0])
confusion_validate = confusion_matrix(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), labels=[1, 0])
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 3))
display1 = ConfusionMatrixDisplay(confusion_matrix=confusion_train, display_labels=[1, 0])
display2 = ConfusionMatrixDisplay(confusion_matrix=confusion_validate, display_labels=[1, 0])
display1.plot(ax=ax[0])
display2.plot(ax=ax[1])
ax[0].set_title("In-sample Performance")
ax[1].set_title("Out-sample Performance")
plt.show()
```

[116] ✓ 1.4s

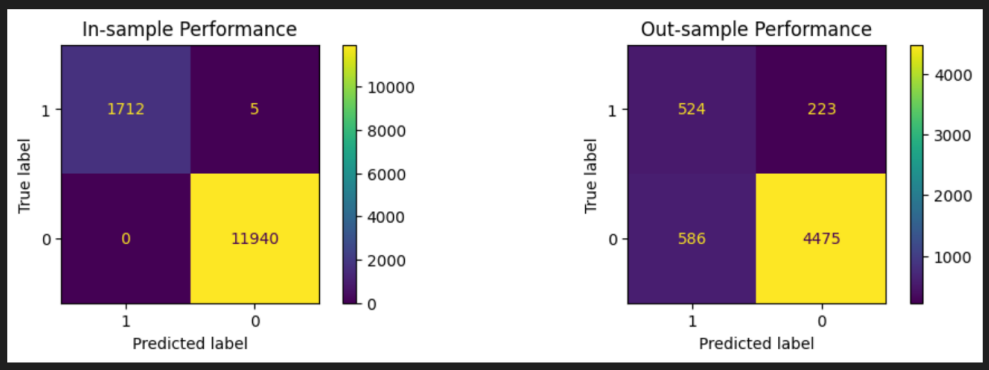


Figure 24. Confusion matrix of KNN model

```
knn_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_validate[selected_features], Y_validate, plot_chance_level=True)
plt.show()
```

[59] ✓ 0.6s

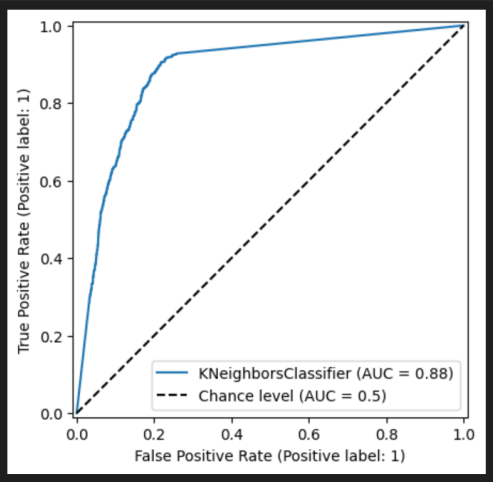


Figure 25a. ROC and AUC of KNN model (validation)

```
knn_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_train[selected_features], Y_train, plot_chance_level=True)
plt.show()
```

[60] ✓ 1.2s

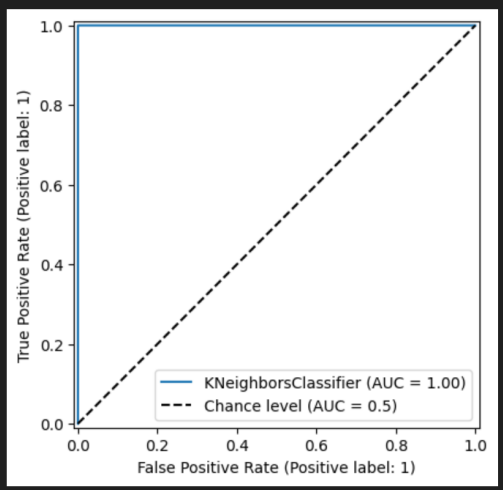


Figure 25b. ROC and AUC of KNN model (training)

```
report = classification_report(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), output_dict=True)
pd.DataFrame(report)
```

[118] ✓ 0.4s

	0	1	accuracy	macro avg	weighted avg
precision	0.952533	0.472072	0.860709	0.712303	0.890738
recall	0.884213	0.701473	0.860709	0.792843	0.860709
f1-score	0.917102	0.564351	0.860709	0.740727	0.871733
support	5061.000000	747.000000	0.860709	5808.000000	5808.000000

Figure 26a. Classification report of KNN model (validation)

```
report = classification_report(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), output_dict=True)
pd.DataFrame(report)
```

[207] ✓ 1.6s

	0	1	accuracy	macro avg	weighted avg
precision	0.999581	1.000000	0.999634	0.999791	0.999634
recall	1.000000	0.997088	0.999634	0.998544	0.999634
f1-score	0.999791	0.998542	0.999634	0.999166	0.999634
support	11940.000000	1717.000000	0.999634	13657.000000	13657.000000

Figure 26b. Classification report of KNN model (training)

- **Gradient Boosting:**

```
param_grid = {'loss' : ['log_loss'],
              'min_samples_leaf' : [1, 2, 3, 4, 5],
              'max_depth' : [20, 15, 10]}

model = HistGradientBoostingClassifier(random_state=0)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(new_predictor[selected_features], new_response)
print(grid_search.best_params_)
print(grid_search.best_score_)
grid_search.best_estimator_
```

[136] ✓ 19.4s

```
Fitting 10 folds for each of 15 candidates, totalling 150 fits
{'loss': 'log_loss', 'max_depth': 15, 'min_samples_leaf': 3}
0.9237437185929649
```

HistGradientBoostingClassifier

```
HistGradientBoostingClassifier(max_depth=15, min_samples_leaf=3, random_state=0)
```

Figure 27. Confusion matrix of gradient boosting model

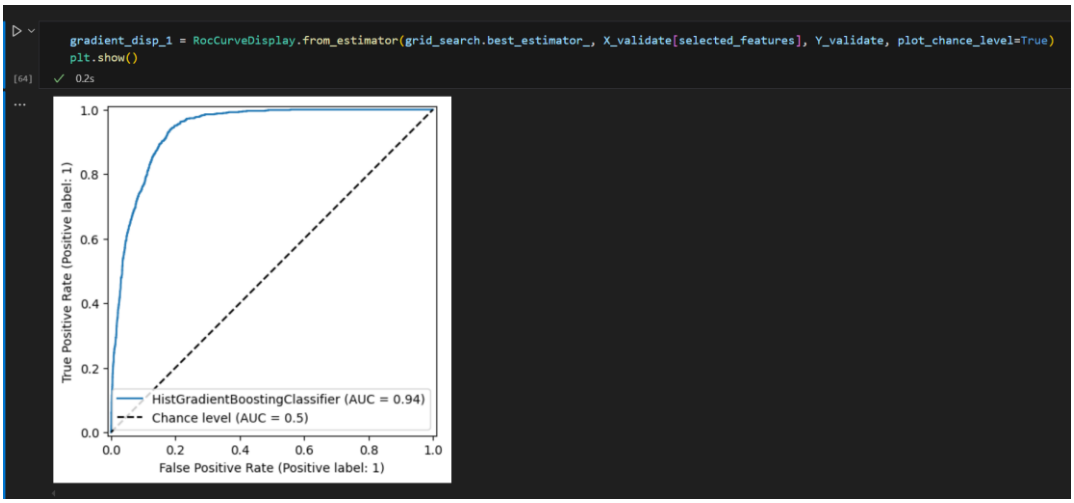


Figure 28a. ROC and AUC of gradient boosting model (validation)

```
gradient_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_train[selected_features], Y_train, plot_chance_level=True)
plt.show()
```
[65] ✓ 0.2s

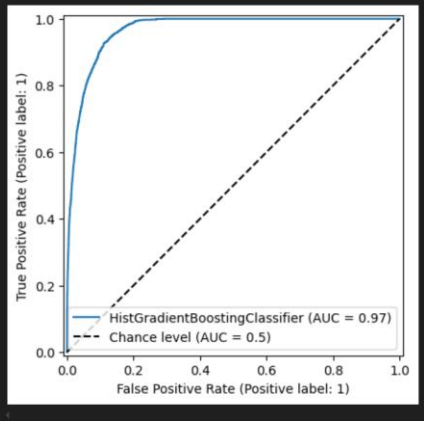


Figure 28b. ROC and AUC of gradient boosting model (training)

```
report = classification_report(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), output_dict=True)
pd.DataFrame(report)
```
[152] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.967333	0.515152	0.87741	0.741242	0.909175
recall	0.889350	0.796519	0.87741	0.842935	0.877410
f1-score	0.926704	0.625657	0.87741	0.776180	0.887984
support	5061.000000	747.000000	0.87741	5808.000000	5808.000000

Figure 29a. Classification report of gradient boosting model (validation)

```
report = classification_report(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), output_dict=True)
pd.DataFrame(report)
```
[210] ✓ 0.1s

	0	1	accuracy	macro avg	weighted avg
precision	0.982402	0.578428	0.904518	0.780415	0.931613
recall	0.907035	0.887012	0.904518	0.897024	0.904518
f1-score	0.943215	0.700230	0.904518	0.821723	0.912667
support	11940.000000	1717.000000	0.904518	13657.000000	13657.000000

Figure 29b. Classification report of gradient boosting model (training)

The Gradient Boosting model's performance is very similar to the Random Forest model's performance. The Kaggle score of this model is 0.62414.

- **Neural Network:**

```
param_grid = {'activation' : ['identity', 'logistic', 'tanh', 'relu'],
              'solver' : ['lbfgs', 'sgd', 'adam']}

model = MLPClassifier(random_state=0, max_iter=300, hidden_layer_sizes=(100))
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(new_predictor[selected_features], new_response)
print(grid_search.best_params_)
print(grid_search.best_score_)
grid_search.best_estimator_
```

[46] ✓ 19m 34.3s

```
Fitting 10 folds for each of 12 candidates, totalling 120 fits
{'activation': 'relu', 'solver': 'adam'}
0.8940536013400335
c:\Users\X1 Carbon\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\neural_network\_m
  warnings.warn(
```

MLPClassifier

MLPClassifier(hidden_layer_sizes=100, max_iter=300, random_state=0)

```
confusion_train = confusion_matrix(new_response, grid_search.best_estimator_.predict(new_predictor[selected_features]), labels=[1,0])
confusion_validate = confusion_matrix(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), labels=[1, 0])
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 3))
display1 = ConfusionMatrixDisplay(confusion_matrix=confusion_train, display_labels=[1, 0])
display2 = ConfusionMatrixDisplay(confusion_matrix=confusion_validate, display_labels=[1, 0])
display1.plot(ax=ax[0])
display2.plot(ax=ax[1])
ax[0].set_title("In-sample Performance")
ax[1].set_title("Out-sample Performance") #450 4768 (SMOTE + no weight)      641 4318 do the selected!!
plt.show()
```

[47] ✓ 0.2s

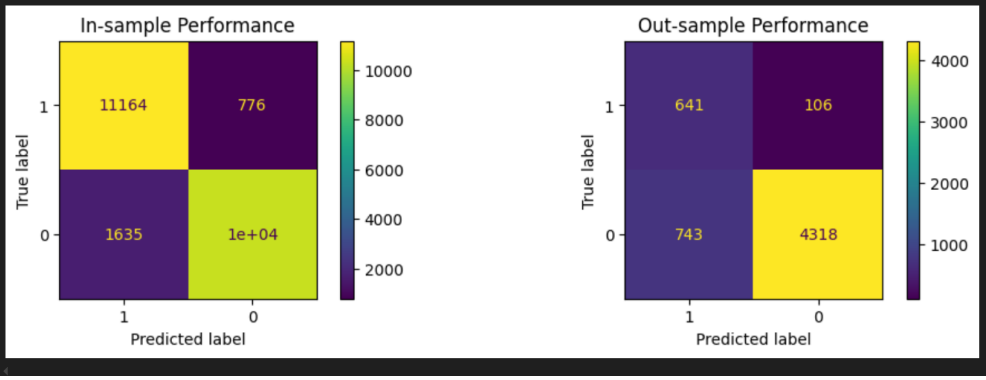


Figure 30. Confusion matrix of neural network model

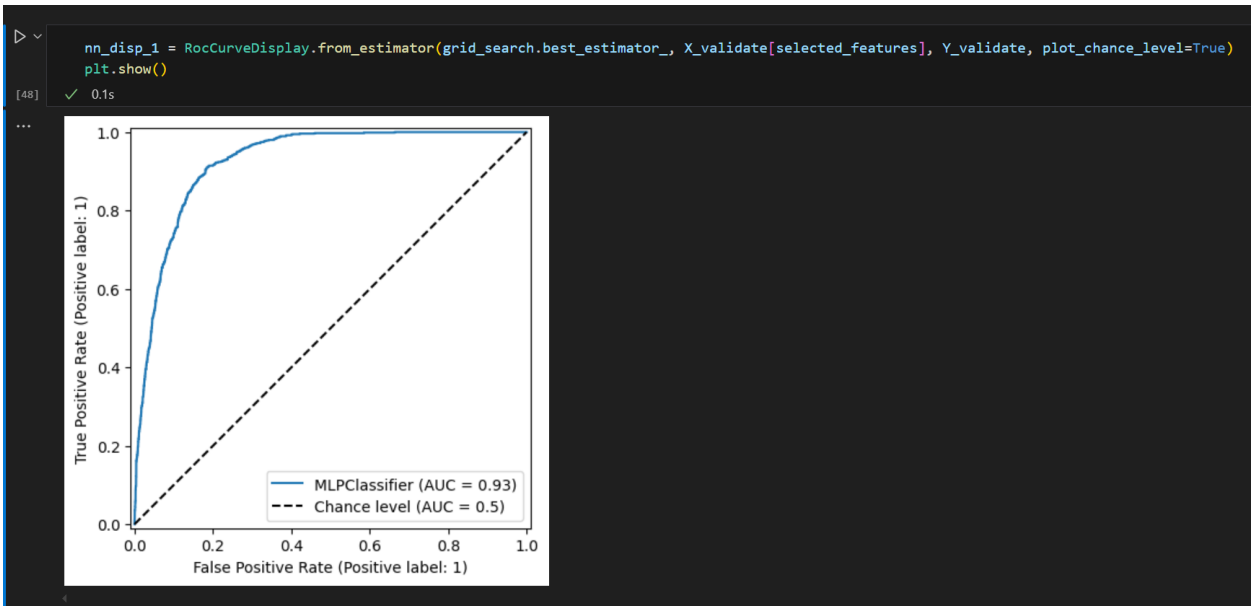


Figure 31a. ROC and AUC of neural network model (validation)

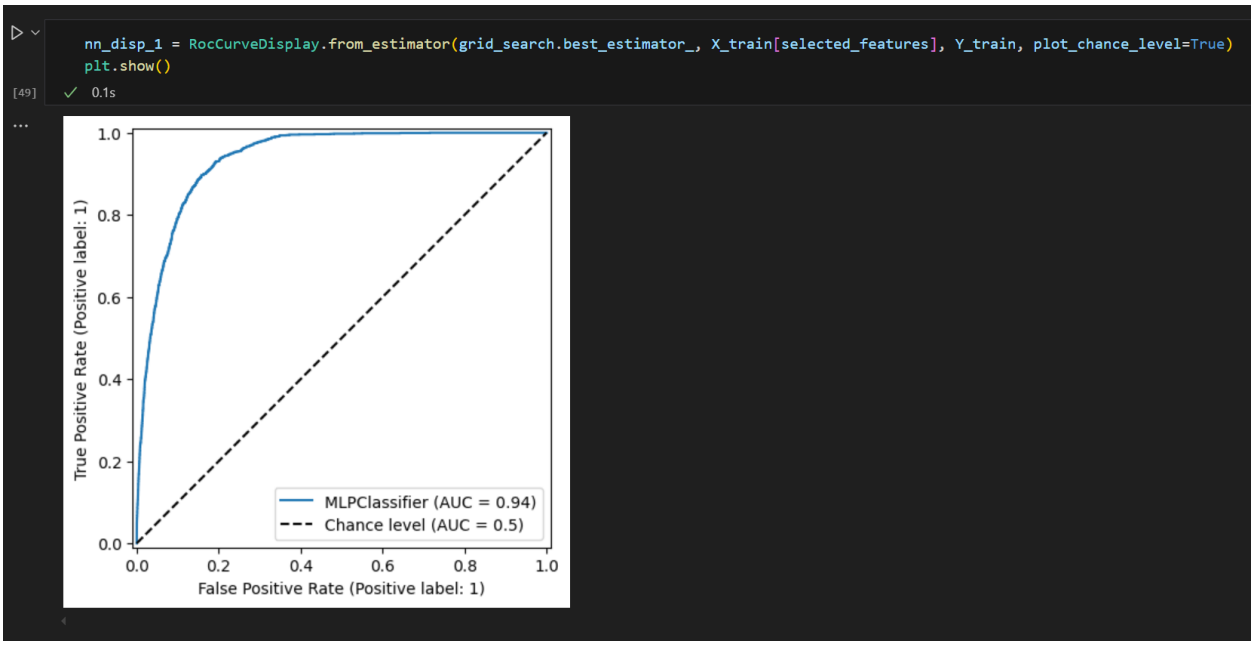


Figure 31b. ROC and AUC of neural network model (training)

```
report = classification_report(Y_validate, grid_search.best_estimator_.predict(X_validate[selected_features]), output_dict=True)
pd.DataFrame(report)
```
[50] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.976040	0.463150	0.853822	0.719595	0.910074
recall	0.853191	0.858099	0.853822	0.855645	0.853822
f1-score	0.910490	0.601595	0.853822	0.756043	0.870762
support	5061.000000	747.000000	0.853822	5808.000000	5808.000000

Figure 32a. Classification report of neural network model (validation)

```
report = classification_report(Y_train, grid_search.best_estimator_.predict(X_train[selected_features]), output_dict=True)
pd.DataFrame(report)
```
[213] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.978168	0.476297	0.86344	0.727233	0.915071
recall	0.863065	0.866045	0.86344	0.864555	0.863440
f1-score	0.917019	0.614590	0.86344	0.765804	0.878997
support	11940.000000	1717.000000	0.86344	13657.000000	13657.000000

Figure 32b. Classification report of neural network model (training)

The neural network is the more complicated model where the two main parameters that I am interested in and modified are $\max\_\text{iter}$ and $hidden\_layer\_sizes$. Initially, the default value of the $\max\ \_iter$ parameter is 100, but after some tunning, I noticed that increasing that increasing the $\max\_iter$ improves the model's performance, where $\max\_iter\ = 300$ is the cut-off. Similiarly, the default value of the $hidden\_layer\_sizes$ is just a 1 element tuple $(100,)$ (1 hidden layer with 100 nodes). I tried various combinations for the number of layers such as $(100, 100, 100)$ and $(50, 50)$. Ultimately, I found the difference in performance to be miniscule. Therefore, I decided not to tamper with $hidden\_layer\_sizes$ and just leave it at the default value.The Kaggle score of this model is 0.58571.

- **Support Vector Machine (SVM):**

SVM

```
param_grid = {'C' : [1, 0.5, 0.05, 0.01]}
w_train = compute_sample_weight(class_weight='balanced', y=Y_train)
class_weight_dict = dict(zip(response.unique(), w_train))

model = SVC(random_state=0)
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)
grid_search.fit(new_predictor, new_response)
print(grid_search.best_params_)
print(grid_search.best_score_)
grid_search.best_estimator_
```

[60] ✓ 10m 22.2s

```
Fitting 10 folds for each of 4 candidates, totalling 40 fits
{'C': 1}
0.903643216080402
```

SVC(C=1, random_state=0)

```
confusion_train = confusion_matrix(Y_train, grid_search.best_estimator_.predict(X_train), labels=[1,0])
confusion_validate = confusion_matrix(Y_validate, grid_search.best_estimator_.predict(X_validate), labels=[1, 0])
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 3))
display1 = ConfusionMatrixDisplay(confusion_matrix=confusion_train, display_labels=[1, 0])
display2 = ConfusionMatrixDisplay(confusion_matrix=confusion_validate, display_labels=[1, 0])
display1.plot(ax=ax[0])
display2.plot(ax=ax[1])
ax[0].set_title("In-sample Performance")
ax[1].set_title("Out-sample Performance") #233 4900 (SMOTE)    | 569 4181 (weight + no SMOTE)    155 4978 (no weight + no SMOTE)
plt.show()
```

[61] ✓ 18.0s

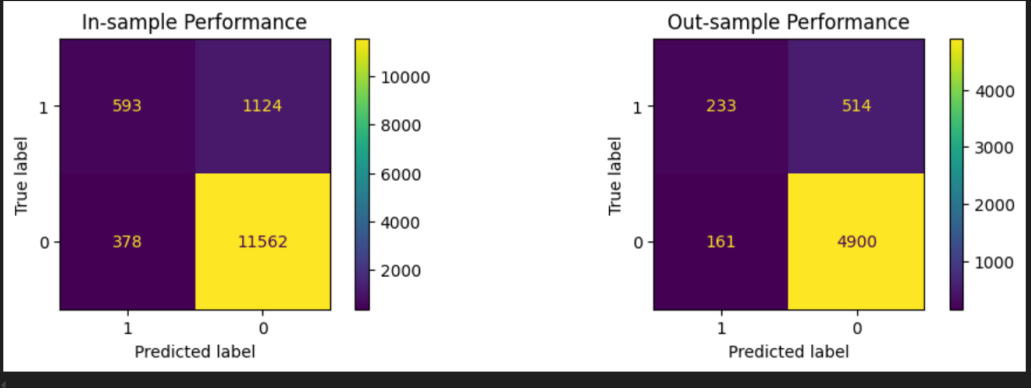


Figure 33. Confusion matrix of SVM model

```
svm_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_validate, Y_validate, plot_chance_level=True)
plt.show()
```

[63] ✓ 5.5s

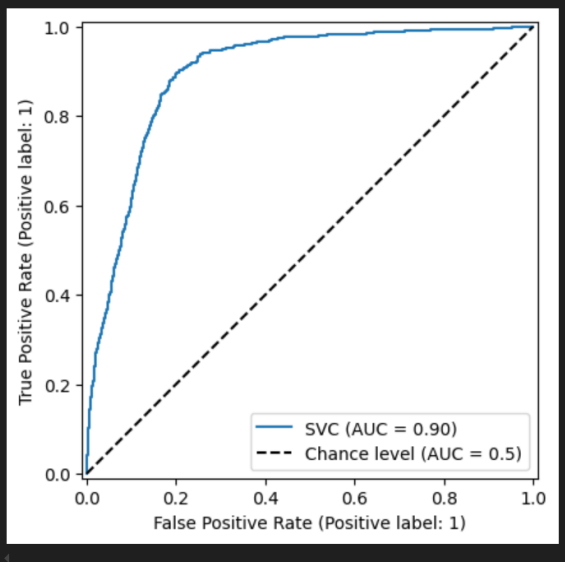


Figure 34a. ROC and AUC of SVM model (validation)

```
svm_disp_1 = RocCurveDisplay.from_estimator(grid_search.best_estimator_, X_train, Y_train, plot_chance_level=True)
plt.show()
```

[64] ✓ 13.5s

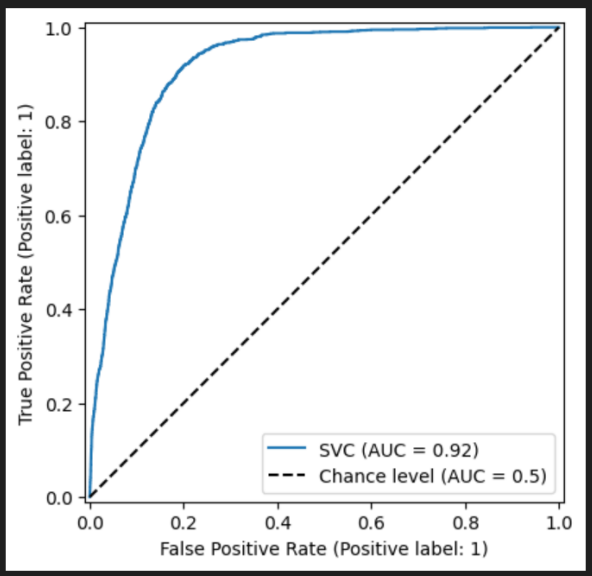


Figure 34b. ROC and AUC of SVM model (training)

```
report = classification_report(Y_validate, grid_search.best_estimator_.predict(X_validate), output_dict=True)
pd.DataFrame(report)
```

[66] ✓ 4.9s

	0	1	accuracy	macro avg	weighted avg
precision	0.905061	0.591371	0.883781	0.748216	0.864715
recall	0.968188	0.311914	0.883781	0.640051	0.883781
f1-score	0.935561	0.408414	0.883781	0.671987	0.867761
support	5061.000000	747.000000	0.883781	5808.000000	5808.000000

Figure 35a. Classification report of SVM model (validation)

```
report = classification_report(Y_train, grid_search.best_estimator_.predict(X_train), output_dict=True)
pd.DataFrame(report)
```

[215] ✓ 12.2s

	0	1	accuracy	macro avg	weighted avg
precision	0.911398	0.610711	0.89002	0.761054	0.873595
recall	0.968342	0.345370	0.89002	0.656856	0.890020
f1-score	0.939008	0.441220	0.89002	0.690114	0.876424
support	11940.000000	1717.000000	0.89002	13657.000000	13657.000000

Figure 35b. Classification report of SVM model (training)

Since the SVM model takes a significant amount of time to train, therefore I did not perform feature selection on this model as the model will be trained on all features. For the SVM model there are 2 parameters that I wanted to modify which are $C\ (Regularization\ parameter)$ and especially $kernel$. However, I decided that testing out all the kernels is not a feasible approach some of the kernels takes up to exponential time which is not suitable for such a large data set. Therefore, I kept the default kernel $(kernel = rbf)$. The SVM might potentially perform better for some other kernels. To avoid overfitting, I tuned the regularization parameter C on a range of descending values. The Kaggle score for this model is 0.43405.

- ## Logistic Regression:

  For the logistic regression, instead of normalizing the data as I have done for all the models above, I will now standardize the data since it is more optimal for models where there is the assumption of linearity. In the case of logistic regression, the model assumes that there is a linear relationship between the independent

variables and log-odds since the logistic regression equation can be more formally written as:

$$\ln\left(\frac{\hat{p}}{1-\hat{p}}\right) = B_0 + B_1 * X_1 + B_2 * X_2 + \cdots + B_n * X_n$$

Therefore, I first tested to see which of the features has a linear relationship with the log-odds as seen in the figure below. I will be building the logistic regression using the statsmodel library instead of using the LogisticRegression class in scikit-learn. This is because statsmodel gives more detailed information, as well as allowing more flexibility.

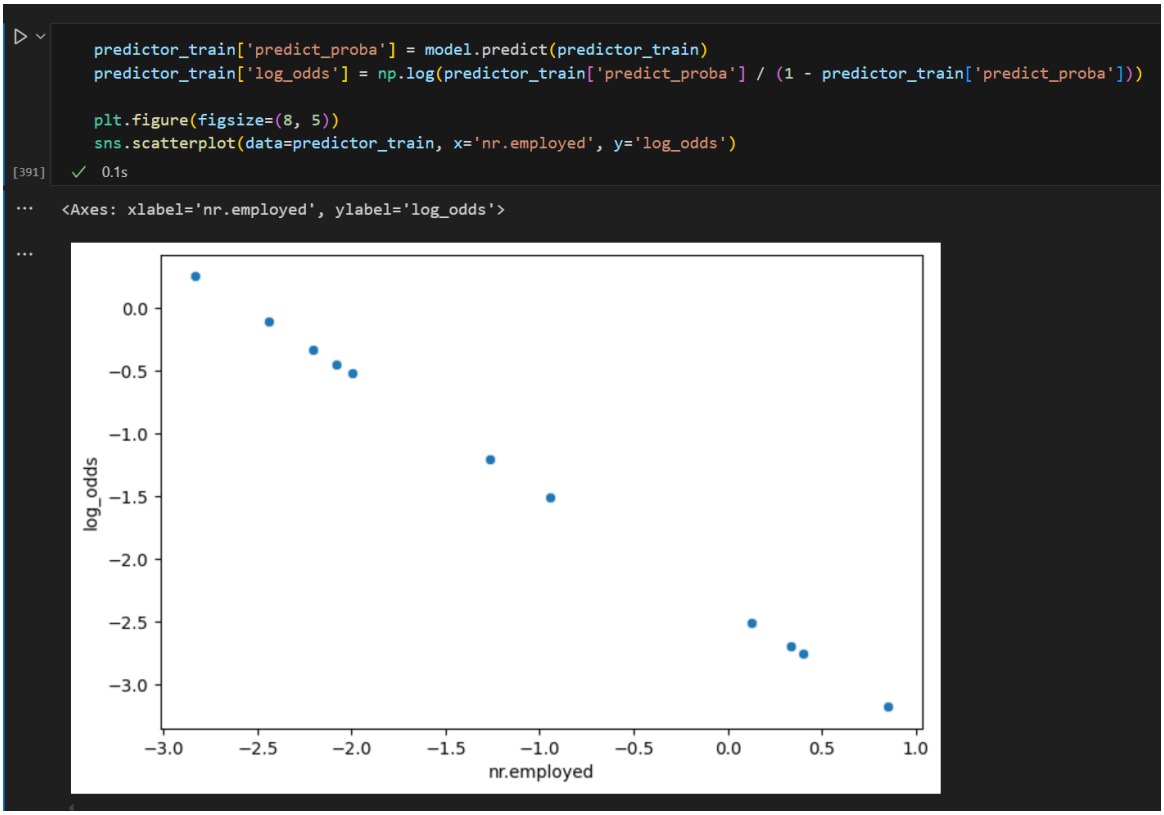


Figure 36. Examining linear relationship between a feature and log-odds

After examining the linear relationship between the features and log-odds, the following potential candidates are:

- nr.employed
- euribor3m
- cons.conf.idx

- cons.price.idx
- emp.var.rate
- previous
- pdays
- campaign
- duration
- age
- education
- day_of_week
- month
- loan
- housing

Now we will build a logistic regression model using these features, however, this will not be our final model, since I will be expecting there to be a problem of multicollinearity.

```
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:             subscribed   No. Observations:                13657
Model:                          Logit   Df Residuals:                    13641
Method:                           MLE   Df Model:                           15
Date:                Fri, 16 May 2025   Pseudo R-squ.:                 0.3680
Time:                        18:23:52   Log-Likelihood:                -3264.4
converged:                       True   LL-Null:                       -5164.7
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const             -2.7809      0.120    -23.200      0.000      -3.016      -2.546
nr.employed       -1.4155      0.210     -6.737      0.000      -1.827      -1.004
euribor3m          1.6194      0.285      5.677      0.000       1.060       2.179
cons.conf.idx     -0.0916      0.048     -1.913      0.056      -0.185       0.002
cons.price.idx     0.0984      0.105      0.935      0.350      -0.108       0.305
emp.var.rate      -1.3838      0.171     -8.085      0.000      -1.719      -1.048
previous          -0.1533      0.030     -5.102      0.000      -0.212      -0.094
pdays             -0.3458      0.025    -13.662      0.000      -0.395      -0.296
campaign          -0.1094      0.054     -2.033      0.042      -0.215      -0.004
duration           1.1887      0.033     36.179      0.000       1.124       1.253
age                0.0209      0.029      0.719      0.472      -0.036       0.078
education          0.0705      0.017      4.150      0.000       0.037       0.104
day_of_week        0.0821      0.024      3.477      0.001       0.036       0.128
month             -0.1260      0.015     -8.557      0.000      -0.155      -0.097
loan              -0.0311      0.090     -0.344      0.731      -0.208       0.146
housing           -0.1313      0.065     -2.007      0.045      -0.260      -0.003
==============================================================================
```

Figure 37. Summary of model built on all selected features

In logistic regression, there is a problem of multicollinearity where features are linearly dependent on each other, which is undesirable and will detrimentally affect the model's performance. Multicollinearity is often detected by examining the Variance Inflation Factor (VIF). There are three cut-offs to VIF, which can be summarized as follows:

- $VIF \geq 10$: High multicollinearity
- $5 \leq VIF < 10$: Moderate multicollinearity
- $VIF < 5$: No mulicollinearity

The VIF for all the selected features can be seen in the output below. As expected, there is indeed a problem of multicollinearity in our model that needs to be resolve.

```
VIF_data = pd.DataFrame({"Feature": predictor_train.columns})
VIF_data['VIF'] = [variance_inflation_factor(predictor_train.values, i) for i in range(predictor_train.shape[1])]
VIF_data.sort_values(by=['VIF'], ascending=False)
```

[394] ✓ 0.1s

	Feature	VIF
2	euribor3m	92.158596
1	nr.employed	42.095622
5	emp.var.rate	39.246354
0	const	14.050957
4	cons.price.idx	7.003834
3	cons.conf.idx	3.070447
6	previous	1.732976
7	pdays	1.607811
13	month	1.564123
10	age	1.053674
11	education	1.052067
8	campaign	1.041964
15	housing	1.012764
12	day_of_week	1.010501
9	duration	1.010470
14	loan	1.003071

Figure 38. VIF of all selected features

We will fix this problem by iteratively removing the feature with the highest VIF. After removing all the features with a VIF above the threshold, the remaining features can be seen below. Note that we do not remove the constant since that is the y-intercept of our regression equation.

```
VIF_data = pd.DataFrame({"Feature": predictor_train.columns})
VIF_data['VIF'] = [variance_inflation_factor(predictor_train.values, i) for i in range(predictor_train.shape[1])]
VIF_data.sort_values(by=['VIF'], ascending=False)
```

[399] ✓ 0.1s

	Feature	VIF
0	const	12.221086
1	nr.employed	1.822744
4	previous	1.730459
5	pdays	1.607275
3	cons.price.idx	1.396868
2	cons.conf.idx	1.074660
11	month	1.072223
8	age	1.050696
9	education	1.044505
6	campaign	1.035992
13	housing	1.011608
7	duration	1.010009
10	day_of_week	1.008727
12	loan	1.002871

Figure 39. Resolving multicollinearity

```
Optimization terminated successfully.
         Current function value: 0.241704
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:             subscribed   No. Observations:                13657
Model:                          Logit   Df Residuals:                    13643
Method:                           MLE   Df Model:                           13
Date:                Fri, 16 May 2025   Pseudo R-squ.:                  0.3609
Time:                        18:36:06   Log-Likelihood:                -3301.0
converged:                       True   LL-Null:                       -5164.7
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const             -2.9080      0.114    -25.483      0.000      -3.132      -2.684
nr.employed       -1.0429      0.035    -29.517      0.000      -1.112      -0.974
cons.conf.idx      0.0777      0.026      2.977      0.003       0.027       0.129
cons.price.idx    -0.0982      0.031     -3.152      0.002      -0.159      -0.037
previous          -0.1654      0.030     -5.514      0.000      -0.224      -0.107
pdays             -0.3427      0.025    -13.611      0.000      -0.392      -0.293
campaign          -0.1314      0.054     -2.440      0.015      -0.237      -0.026
duration           1.1789      0.033     36.124      0.000       1.115       1.243
age                0.0255      0.029      0.882      0.378      -0.031       0.082
education          0.0740      0.017      4.382      0.000       0.041       0.107
day_of_week        0.0779      0.023      3.322      0.001       0.032       0.124
month             -0.0917      0.012     -7.415      0.000      -0.116      -0.067
loan              -0.0443      0.090     -0.493      0.622      -0.220       0.132
housing           -0.1297      0.065     -1.992      0.046      -0.257      -0.002
==============================================================================
```

Figure 37. Summary of model after resolving multicollinearity problem

For the final fine-tuning step, we need to examine the significance of each feature and remove features that are insignificant, i.e., not contributing much to the log-odds. To do this, we will be using the 0.05 confidence level, meaning if a feature's $p - value$ (highlighted red in the figure above) is greater than 0.05, we will remove it from the model. Note that we remove the insignificant features iteratively / one by one, and not in one go.

```
Optimization terminated successfully.
         Current function value: 0.241888
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:             subscribed   No. Observations:                13657
Model:                          Logit   Df Residuals:                    13646
Method:                           MLE   Df Model:                           10
Date:                Fri, 16 May 2025   Pseudo R-squ.:                  0.3604
Time:                        19:52:16   Log-Likelihood:                -3303.5
converged:                       True   LL-Null:                       -5164.7
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const            -2.9637      0.108    -27.320      0.000      -3.176      -2.751
nr.employed      -1.0435      0.035    -29.655      0.000      -1.112      -0.975
cons.conf.idx     0.0845      0.026      3.294      0.001       0.034       0.135
cons.price.idx   -0.0955      0.031     -3.068      0.002      -0.156      -0.034
previous         -0.1656      0.030     -5.529      0.000      -0.224      -0.107
pdays            -0.3424      0.025    -13.611      0.000      -0.392      -0.293
campaign         -0.1301      0.054     -2.420      0.016      -0.236      -0.025
duration          1.1793      0.033     36.139      0.000       1.115       1.243
education         0.0701      0.017      4.231      0.000       0.038       0.103
day_of_week       0.0767      0.023      3.272      0.001       0.031       0.123
month            -0.0927      0.012     -7.508      0.000      -0.117      -0.069
==============================================================================
```

Figure 38. Summary of model after removing insignificant features

Notice that despite the $p - value$ of the feature ‘housing’ is technically still smaller than 0.05 but I still chose to remove it since it is borderline to the threshold. Additionally, removing the feature also improves the performance of the model slightly.

```
confusion_train = confusion_matrix(Y_train, list(map(round, model.predict(predictor_train))), labels=[1,0])
confusion_validate = confusion_matrix(Y_validate, list(map(round, model.predict(predictor_validate))), labels=[1, 0])
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 3))
display1 = ConfusionMatrixDisplay(confusion_matrix=confusion_train, display_labels=[1, 0])
display2 = ConfusionMatrixDisplay(confusion_matrix=confusion_validate, display_labels=[1, 0])
display1.plot(ax=ax[0])
display2.plot(ax=ax[1])
ax[0].set_title("In-sample Performance")
ax[1].set_title("Out-sample Performance")
plt.show()
```

[593] ✓ 0.2s

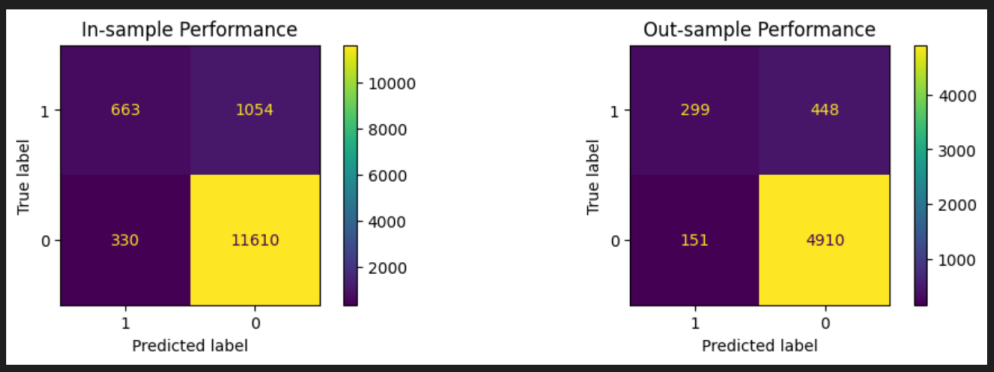


Figure 39. Confusion matrix of logistic regression model

```
fpr, tpr, thresholds = roc_curve(Y_validate, model.predict(predictor_validate))
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, label=f'Logistic Regression (AUC = {roc_auc:.5f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Chance level (AUC = 0.5)')
plt.xlabel('False Positive Rate (Positive label: 1)')
plt.ylabel('True Positive Rate (Positive label: 1)')
plt.legend()
plt.show()
```

[594] ✓ 0.1s

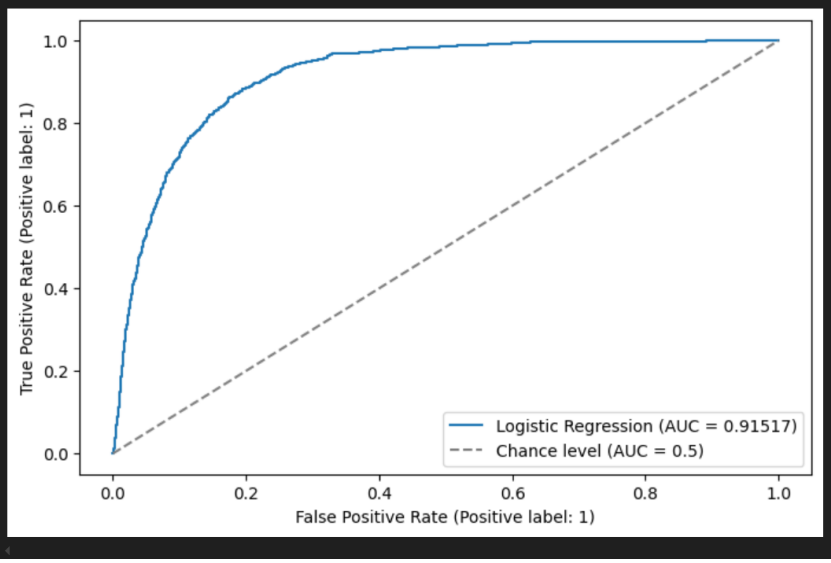


Figure 40a. ROC and AUC of logistic regression model (validation)

```
fpr, tpr, thresholds = roc_curve(Y_train, model.predict(predictor_train))
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, label=f'Logistic Regression (AUC = {roc_auc:.5f})')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Chance level (AUC = 0.5)')
plt.xlabel('False Positive Rate (Positive label: 1)')
plt.ylabel('True Positive Rate (Positive label: 1)')
plt.legend()
plt.show()
```

[595] ✓ 0.1s

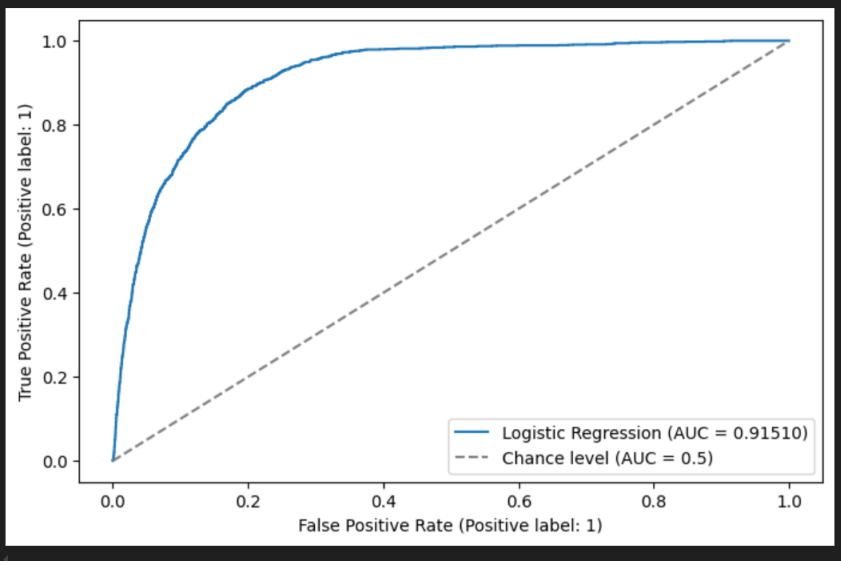


Figure 40b. ROC and AUC of logistic regression model (training)

```
report = classification_report(Y_validate, list(map(round, model.predict(predictor_validate))), output_dict=True)
pd.DataFrame(report)
```

[596] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.916387	0.664444	0.896866	0.790416	0.883983
recall	0.970164	0.400268	0.896866	0.685216	0.896866
f1-score	0.942509	0.499582	0.896866	0.721046	0.885542
support	5061.000000	747.000000	0.896866	5808.000000	5808.000000

Figure 41a. Classification report of logistic regression model (validation)

```
report = classification_report(Y_train, list(map(round, model.predict(predictor_train))), output_dict=True)
pd.DataFrame(report)
```

[589] ✓ 0.0s

	0	1	accuracy	macro avg	weighted avg
precision	0.916772	0.667674	0.89866	0.792223	0.885455
recall	0.972362	0.386139	0.89866	0.679250	0.898660
f1-score	0.943749	0.489299	0.89866	0.716524	0.886614
support	11940.000000	1717.000000	0.89866	13657.000000	13657.000000

Figure 41b. Classification report of logistic regression model (training)

The Kaggle score for this model is 0.50. Despite the performance of this model is only as good as a random guess, this model has the highest precision for class 1 out of all the models in the validation data set. Furthermore, the logistic regression model has the highest accuracy amongst all the other models in the validation set (See section 6). Because all of the model's features are significant, meaning they do affect the probability of whether the customer subscribes to a term deposit, this means the logistic regression model is potentially viable. We are just missing something else to fully capture the relationship. This can be interaction terms or additional information / features.

6. Summary of all models' result

	Training								Validating							
	Accuracy	Recall_0	Recall_1	Precision_0	Precision_1	F1 Score_0	F1 Score_1	AUC	Accuracy	Recall_0	Recall_1	Precision_0	Precision_1	F1 Score_0	F1 Score_1	AUC
DecisionTree	0.95	0.94	0.998	0.999	0.72	0.97	0.84	0.99	0.86	0.89	0.64	0.94	0.47	0.92	0.54	0.77
RandomForest	0.997	0.997	0.997	0.999	0.98	0.998	0.99	1.00	0.89	0.91	0.74	0.96	0.55	0.93	0.63	0.94
GradientBoosting	0.90	0.91	0.89	0.98	0.58	0.94	0.70	0.97	0.88	0.89	0.80	0.97	0.52	0.93	0.63	0.94
KNN	0.999	1.00	0.997	0.999	1.00	0.999	0.998	1.00	0.86	0.88	0.70	0.95	0.47	0.92	0.56	0.88
SVM	0.89	0.97	0.35	0.91	0.61	0.94	0.44	0.92	0.88	0.97	0.31	0.91	0.59	0.94	0.41	0.90
LogisticRegression	0.90	0.97	0.39	0.92	0.67	0.94	0.49	0.92	0.90	0.97	0.40	0.92	0.66	0.94	0.50	0.92
NeuralNetwork	0.86	0.86	0.87	0.98	0.48	0.92	0.61	0.94	0.85	0.85	0.86	0.98	0.46	0.91	0.60	0.93

The table above summarizes the performance of all classifiers in both the training and validation datasets. We can see that there is the aforementioned problem of overfitting, as the classifiers did extremely well on the training dataset but performed poorly on the validation dataset. Evidently, the classifiers are having difficulties detecting the minority class 1. But leaving aside the overfitting problem, we can see that the KNN and RandomForest models perform relatively good across all performance metrics, excluding the precision of class 1. This indicates that with further tuning to resolve the issue of overfitting and class imbalances, these 2 models might be the best.

However, if we strictly evaluate the model's performance based on accuracy only, LogisticRegression is the best model. LogisticRegression's recall for class 0 and precision for class 1 are better than KNN and RandomForest models. Moreover, the

LogisticRegression model's ability to distinguish between the 2 classes is also very high, as seen from its AUC score. Therefore, despite its Kaggle score indicating it is no better than a random guess, LogisticRegression has potential and is just missing some key features or interaction terms.

7. Testing score of all classifiers on Kaggle

	Kaggle score
DecisionTree	0.57950
RandomForest	0.63088
GradientBoosting	0.62414
KNN	0.58058
SVM	0.43405
LogisticRegression	0.50
NeuralNetwork	0.58571

The table above shows the Kaggle score of all the classifiers. Unsurprisingly, the RandomForest model has the highest performance followed by the GradientBoosting model. Conversely, the SVM model performs the worst. The KNN model, while performing great in the training data set and decent in the validation set turns out to perform worse in the test data set.

8. Best classifier result Kaggle submission and score