Implementation of Random Forest Classification with Scikit-Learn

Dataset URL: https://www.kaggle.com/datasets/yufengsui/portuguese-bank-marketing-data-set

This tutorial explains how to use random forests for classification in Python. We will cover:

- How random forests work?
- How to use them for classification?
- How to evaluate their performance?

While random forests can be used for both classification and regression, this article will focus on building a classification model.

Overview

Random forests are a popular supervised machine learning algorithm.

- Random forests are for supervised machine learning, where there is a labeled target variable.
- Random forests can be used for solving regression (numeric target variable) and classification (categorical target variable) problems.
- Random forests are an ensemble method, meaning they combine predictions from other models.
- Each of the smaller models in the random forest ensemble is a decision tree.

How Random Forest Classification works

Imagine you have a complex problem to solve, and you gather a group of experts from different fields to provide their input. Each expert provides their opinion based on their expertise and experience. Then, the experts would vote to arrive at a final decision.

In a random forest classification, multiple decision trees are created using different random subsets of the data and features.

Each decision tree is like an expert, providing its opinion on how to classify the data.

Predictions are made by calculating the prediction for each decision tree, then taking the most popular result. (For regression, predictions use an averaging technique instead.)

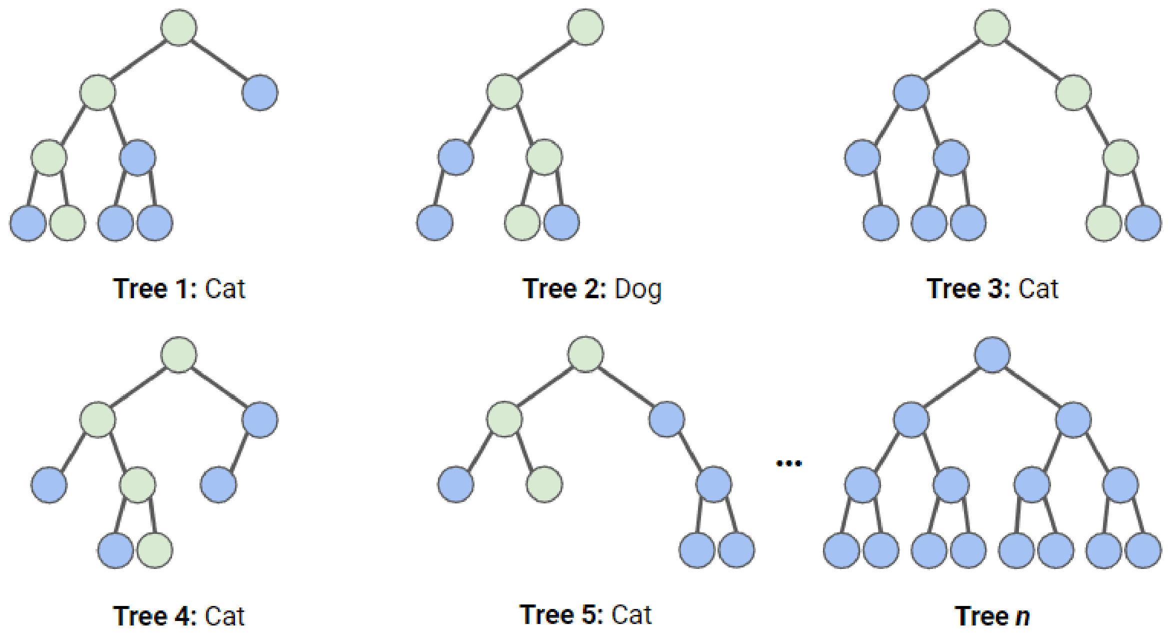
In the diagram below, we have a random forest with n decision trees, and we've shown the first 5, along with their predictions (either "Dog" or "Cat").

Each tree is exposed to a different number of features and a different sample of the original dataset, and as such, every tree can be different.

Each tree makes a prediction. Looking at the first 5 trees, we can see that 4/5 predicted the sample was a Cat.

The green circles indicate a hypothetical path the tree took to reach its decision.

The random forest would count the number of predictions from decision trees for Cat and for Dog, and choose the most popular prediction.



The Dataset

https://www.kaggle.com/datasets/yufengsui/portuguese-bank-marketing-data-set

This dataset consists of direct marketing campaigns by a Portuguese banking institution using phone calls. The campaigns aimed to sell subscriptions to a bank term deposit. We are going to store this dataset in a variable called bank_data.

The columns we will use are:

- age: The age of the person who received the phone call
- default: Whether the person has credit in default
- cons.price.idx: Consumer price index score at the time of the call
- cons.conf.idx:Consumer confidence index score at the time of the call
- y: Whether the person subscribed (this is what we're trying to predict)

Importing Packages

The following packages and functions are used in this tutorial:

```
# Data Processing

import pandas as pd

import numpy as np



# Modelling

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score, confusion_matrix, precision_score,
recall_score, ConfusionMatrixDisplay

from sklearn.model_selection import RandomizedSearchCV, train_test_split

from scipy.stats import randint



# Tree Visualisation

from sklearn.tree import export_graphviz

from IPython.display import Image

import graphviz
```

Explanation:

This code imports various libraries that will be used in the tutorial.

• **pandas** and **numpy** are used for data processing.

• **RandomForestClassifier**, **accuracy_score**, **confusion_matrix**, **precision_score**, **recall_score**, **ConfusionMatrixDisplay**, **RandomizedSearchCV**, **train_test_split**, and **randint** are all from the **sklearn** library and are used for modelling.

• **export_graphviz** is from the **sklearn.tree** library and is used for tree visualisation.

• **Image** and **graphviz** are used to display the tree visualisation in the Jupyter notebook.

Random Forests Workflow

To fit and train this model, we'll be following **The Machine Learning Workflow** infographic; however, as our data is pretty clean, we won't be carrying out every step.

We will do the following:

- Feature engineering
- Split the data
- Train the model
- Hyperparameter tuning
- Assess model performance

Preprocessing Data for Random Forests

Tree-based models are much more robust to outliers than linear models, and they do not need variables to be normalized to work. As such, we need to do very little preprocessing on our data.

- We will map our 'default' column, which contains no and yes, to 0s and 1s, respectively. We will treat unknown values as no for this example.
- We will also map our target, y, to 1s and 0s.

```
bank_data['default'] = bank_data['default'].map({'no':0,'yes':1,'unknown':0})
```

```
bank_data['y'] = bank_data['y'].map({'no':0,'yes':1})
```

Explanation:

This code is written in Python and it is used to map categorical values to numerical values in a Pandas DataFrame called **bank_data**.

• The first line of code maps the values in the 'default' column of the DataFrame to numerical values.

• The dictionary passed to the **map()** function maps 'no' to 0, 'yes' to 1, and 'unknown' to 0.

• This means that all 'no' and 'unknown' values in the 'default' column will be mapped to 0, while all 'yes' values will be mapped to 1.

• The resulting numerical values are then stored in a new column called 'default'.

• The second line of code does the same thing for the 'y' column of the DataFrame.

• The dictionary passed to the **map()** function maps 'no' to 0 and 'yes' to 1.

• This means that all 'no' values in the 'y' column will be mapped to 0, while all 'yes' values will be mapped to 1.

• The resulting numerical values are then stored in a new column called 'y'.

• Overall, this code is useful for converting categorical data into numerical data, which can be easier to work with in certain types of analyses.

Splitting the Data

When training any supervised learning model, it is important to split the data into training and test data. The training data is used to fit the model. The algorithm uses the training data to learn the relationship between the features and the target. The test data is used to evaluate the performance of the model.

The code below splits the data into separate variables for the features and target, then splits into training and test data.

```
# Split the data into features (X) and target (y)

X = bank_data.drop('y', axis=1)

y = bank_data['y']



# Split the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Explanation:

This code is written in Python and it is used to split the data into training and test sets for a machine learning model.

• First, the code separates the features (X) and target (y) variables from the original dataset **bank_data** using the **drop()** method.

• The **drop()** method is used to remove a specified column from the dataset.

• In this case, the column 'y' is removed from the dataset and assigned to the variable X.

• The column 'y' is assigned to the variable y.

• Next, the **train_test_split()** function from the scikit-learn library is used to split the data into training and test sets.

• The function takes four arguments: the features (X), the target (y), the test size (0.2 in this case), and a random state (which is not specified in this code).

• The test size argument specifies the proportion of the data that should be used for testing the model.

• In this case, 20% of the data is used for testing and 80% is used for training.

• The function returns four variables: X_train, X_test, y_train, and y_test.

• These variables contain the training and test sets for the features and target variables.

• The training sets are used to train the machine learning model, and the test sets are used to evaluate the performance of the model.

Fitting and Evaluating the Model

We first create an instance of the Random Forest model, with the default parameters. We then fit this to our training data. We pass both the features and the target variable, so the model can learn.

```
rf = RandomForestClassifier()

rf.fit(X_train, y_train)
```

Explanation:

This code creates an instance of the **RandomForestClassifier** class and assigns it to the variable **rf**.

• Then, it fits the model to the training data **X_train** and **y_train** using the **fit()** method.

• The **RandomForestClassifier** is a machine learning algorithm that creates a forest of decision trees and combines their predictions to make a final prediction.

• The **fit()** method trains the model on the input data by adjusting the parameters of the decision trees to minimize the error between the predicted and actual values.

• Overall, this code trains a random forest classifier on the training data **X_train** and **y_train**.

At this point, we have a trained Random Forest model, but we need to find out whether it is making accurate predictions.

```
y_pred = rf.predict(X_test)
```

Explanation:

This code uses the **predict** method of a **RandomForestClassifier** object (**rf**) to make predictions on a set of test data (**X_test**).

• The predicted values are then stored in the variable **y_pred**.

• This is a common step in machine learning workflows, where the model is trained on a set of training data and then used to make predictions on new, unseen data.

The simplest way to evaluate this model is using accuracy; we check the predictions against the actual values in the test set and count up how many the model got right.

```
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

Explanation:

This code calculates the accuracy of a machine learning model's predictions by comparing the predicted values (**y_pred**) to the actual values (**y_test**) and then using the **accuracy_score** function from the scikit-learn library to calculate the accuracy.

• The resulting accuracy value is then printed to the console using the **print** function.

Output:

```
Accuracy: 0.888
```

Explanation:

This code snippet is not actually code, but rather a result or output of some code.

• It is likely that the code preceding this output was a machine learning model that was trained and tested on some data, and the accuracy of the model was calculated to be 0.888.

• The language used to generate this output is not specified, but it could be any language commonly used for machine learning such as Python, R, or MATLAB.

This is a pretty good score! However, we may be able to do better by optimizing our hyperparameters.

Visualizing the Results

We can use the following code to visualize our first 3 trees.

```
# Export the first three decision trees from the forest



for i in range(3):

    tree = rf.estimators_[i]

    dot_data = export_graphviz(tree,

                               feature_names=X_train.columns,

                               filled=True,
                               max_depth=2,

                               impurity=False,

                               proportion=True)

    graph = graphviz.Source(dot_data)

    display(graph)
```

Explanation:

This code exports the first three decision trees from a random forest model.

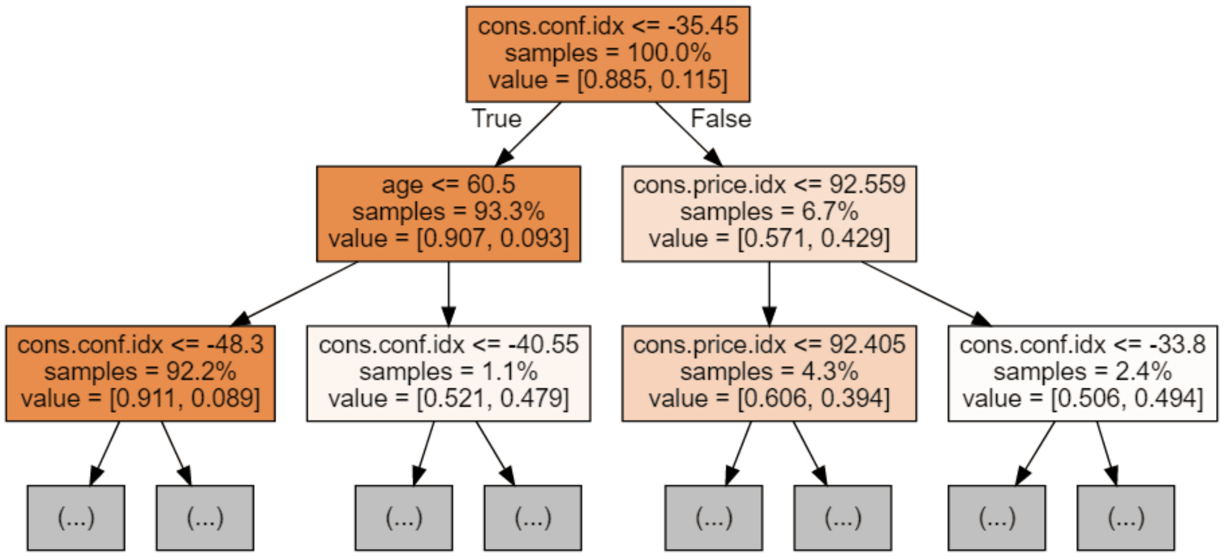
• The **for** loop iterates three times, and for each iteration, it selects the **i**-th decision tree from the **estimators_** attribute of the random forest model (**rf**).

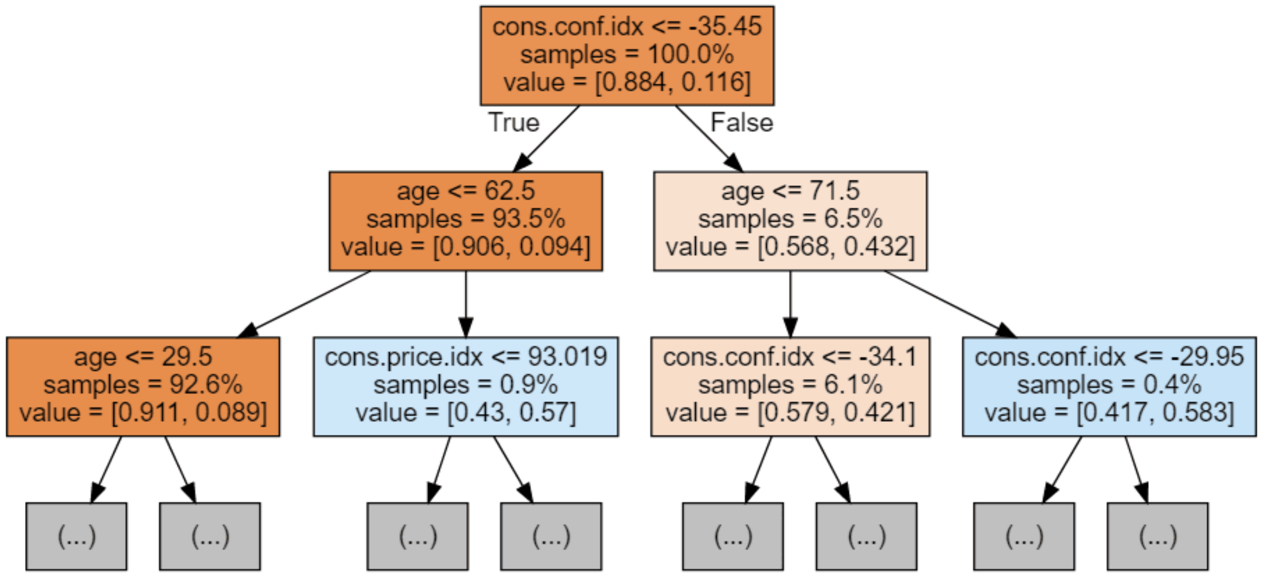
• Then, it uses the **export_graphviz** function from the **sklearn.tree** module to create a Graphviz representation of the decision tree.

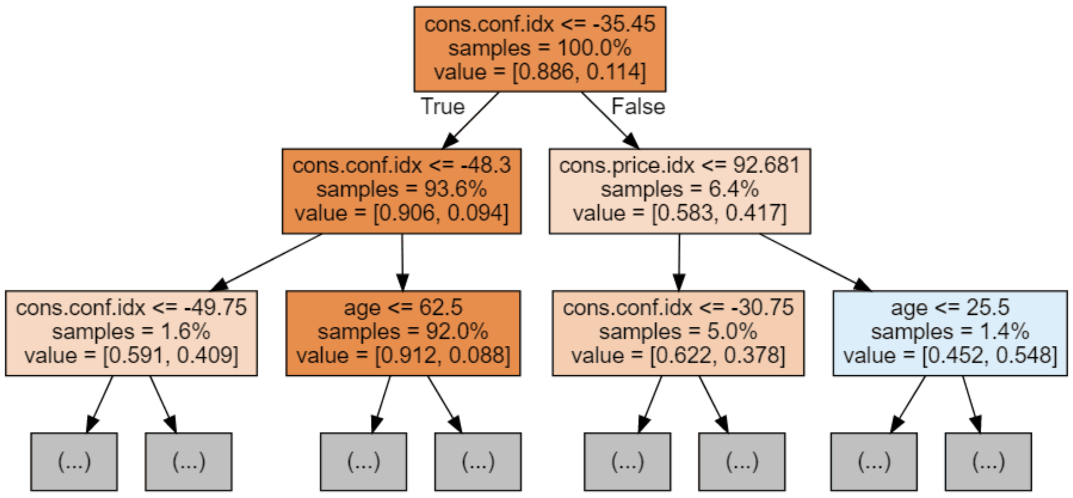
• The function takes several arguments, including the decision tree object (**tree**), the names of the features used in the model (**X_train.columns**), and options for formatting the output (**filled**, **max_depth**, **impurity**, and **proportion**).

• The resulting Graphviz representation is then displayed using the **graphviz.Source** function and the **display** function.

• Overall, this code is useful for visualizing the decision trees in a random forest model, which can help with understanding how the model is making predictions.


cons.conf.idx <= -35.45
samples = 100.0%
value = [0.885, 0.115]
True
False
age <= 60.5
samples = 93.3%
value = [0.907, 0.093]
cons.price.idx <= 92.559
samples = 6.7%
value = [0.571, 0.429]
cons.conf.idx <= -48.3
samples = 92.2%
value = [0.911, 0.089]
cons.conf.idx <= -40.55
samples = 1.1%
value = [0.521, 0.479]
cons.price.idx <= 92.405
samples = 4.3%
value = [0.606, 0.394]
cons.conf.idx <= -33.8
samples = 2.4%
value = [0.506, 0.494]
(...)
(...)
(...)
(...)
(...)
(...)
(...)
(...)


cons.conf.idx <= -35.45
samples = 100.0%
value = [0.884, 0.116]
True
False
age <= 62.5
samples = 93.5%
value = [0.906, 0.094]
age <= 71.5
samples = 6.5%
value = [0.568, 0.432]
age <= 29.5
samples = 92.6%
value = [0.911, 0.089]
cons.price.idx <= 93.019
samples = 0.9%
value = [0.43, 0.57]
cons.conf.idx <= -34.1
samples = 6.1%
value = [0.579, 0.421]
cons.conf.idx <= -29.95
samples = 0.4%
value = [0.417, 0.583]
(...)
(...)
(...)
(...)
(...)
(...)
(...)
(...)


cons.conf.idx <= -35.45
samples = 100.0%
value = [0.886, 0.114]
True
False
cons.conf.idx <= -48.3
samples = 93.6%
value = [0.906, 0.094]
cons.price.idx <= 92.681
samples = 6.4%
value = [0.583, 0.417]
cons.conf.idx <= -49.75
samples = 1.6%
value = [0.591, 0.409]
age <= 62.5
samples = 92.0%
value = [0.912, 0.088]
cons.conf.idx <= -30.75
samples = 5.0%
value = [0.622, 0.378]
age <= 25.5
samples = 1.4%
value = [0.452, 0.548]
(...)
(...)
(...)
(...)
(...)
(...)
(...)
(...)

Each tree image is limited to only showing the first few nodes. These trees can get very large and difficult to visualize. The colors represent the majority class of each node (box, with red indicating majority 0 (no subscription) and blue indicating majority 1 (subscription). The colors get darker the closer the node gets to being fully 0 or 1. Each node also contains the following information:

1. The variable name and value used for splitting
2. The % of total samples in each split
3. The % split between classes in each split

Hyperparameter Tuning

The code below uses Scikit-Learn's **RandomizedSearchCV**, which will randomly search parameters within a range per hyperparameter. We define the hyperparameters to use and their ranges in the param_dist dictionary. In our case, we are using:

- **n_estimators**: the number of decision trees in the forest. Increasing this hyperparameter generally improves the performance of the model but also increases the computational cost of training and predicting.
- **max_depth**: the maximum depth of each decision tree in the forest. Setting a higher value for max_depth can lead to overfitting while setting it too low can lead to underfitting.

```
param_dist = {'n_estimators': randint(50,500),

              'max_depth': randint(1,20)}



# Create a random forest classifier

rf = RandomForestClassifier()



# Use random search to find the best hyperparameters
```

```
rand_search = RandomizedSearchCV(rf,
                                 param_distributions = param_dist,
                                 n_iter=5,
                                 cv=5)


# Fit the random search object to the data
rand_search.fit(X_train, y_train)
```

Explanation:

This code is performing hyperparameter tuning for a random forest classifier using random search.

• First, a dictionary **param_dist** is created with two hyperparameters to tune: **n_estimators** and **max_depth**.

• The values for these hyperparameters are randomly sampled from a uniform distribution between 50 and 500 for **n_estimators** and between 1 and 20 for **max_depth**.

• Next, a **RandomForestClassifier** object is created.

• Then, a **RandomizedSearchCV** object is created with the **RandomForestClassifier** object, the **param_dist** dictionary, and other parameters such as **n_iter** (the number of parameter settings that are sampled) and **cv** (the number of cross-validation folds to use).

• Finally, the **RandomizedSearchCV** object is fit to the training data (**X_train** and **y_train**) to find the best hyperparameters for the random forest classifier.

RandomizedSearchCV will train many models (defined by n_iter_ and save each one as variables, the code below creates a variable for the best model and prints the hyperparameters. In this case, we haven’t passed a scoring system to the function, so it defaults to accuracy. This function also uses cross validation, which means it splits the data into five equal-sized groups and uses 4 to train and 1 to test the result. It will loop through each group and give an accuracy score, which is averaged to find the best model.

```python
# Create a variable for the best model

best_rf = rand_search.best_estimator_



# Print the best hyperparameters

print('Best hyperparameters:',  rand_search.best_params_)
```

Explanation:

This code snippet is written in Python.

• The first line creates a variable called **best_rf** and assigns it the value of the best estimator found by a random search.

• The **rand_search** object is assumed to have been previously defined and used to search for the best hyperparameters for a machine learning model.

• The second line prints out the best hyperparameters found by the random search, which are stored in the **best_params_** attribute of the **rand_search** object.

• Overall, this code is used to retrieve the best model and its hyperparameters after performing a random search.

Output:

```
Best hyperparameters: {'max_depth': 5, 'n_estimators': 260}
```

Explanation:

This code snippet is not written in any specific programming language.

• It is simply displaying the best hyperparameters found during a model training process.

• The hyperparameters are 'max_depth' and 'n_estimators', and their respective values are 5 and 260.

• These hyperparameters were likely found using a grid search or some other optimization technique to find the combination of hyperparameters that resulted in the best model performance.

More Evaluation Metrics

Let’s look at the confusion matrix. This plots what the model predicted against what the correct prediction was. We can use this to understand the tradeoff between false positives (top right) and false negatives(bottom left) We can plot the confusion matrix using this code:

```
# Generate predictions with the best model

y_pred = best_rf.predict(X_test)



# Create the confusion matrix

cm = confusion_matrix(y_test, y_pred)



ConfusionMatrixDisplay(confusion_matrix=cm).plot();
```

Explanation:

This code snippet is used to generate predictions with the best model and create a confusion matrix to evaluate the performance of the model.

• First, the **predict()** method is called on the **best_rf** object with **X_test** as the argument to generate predictions for the test data.

• The predicted values are stored in the **y_pred** variable.

• Next, the **confusion_matrix()** function is called with **y_test** and **y_pred** as arguments to create a confusion matrix.

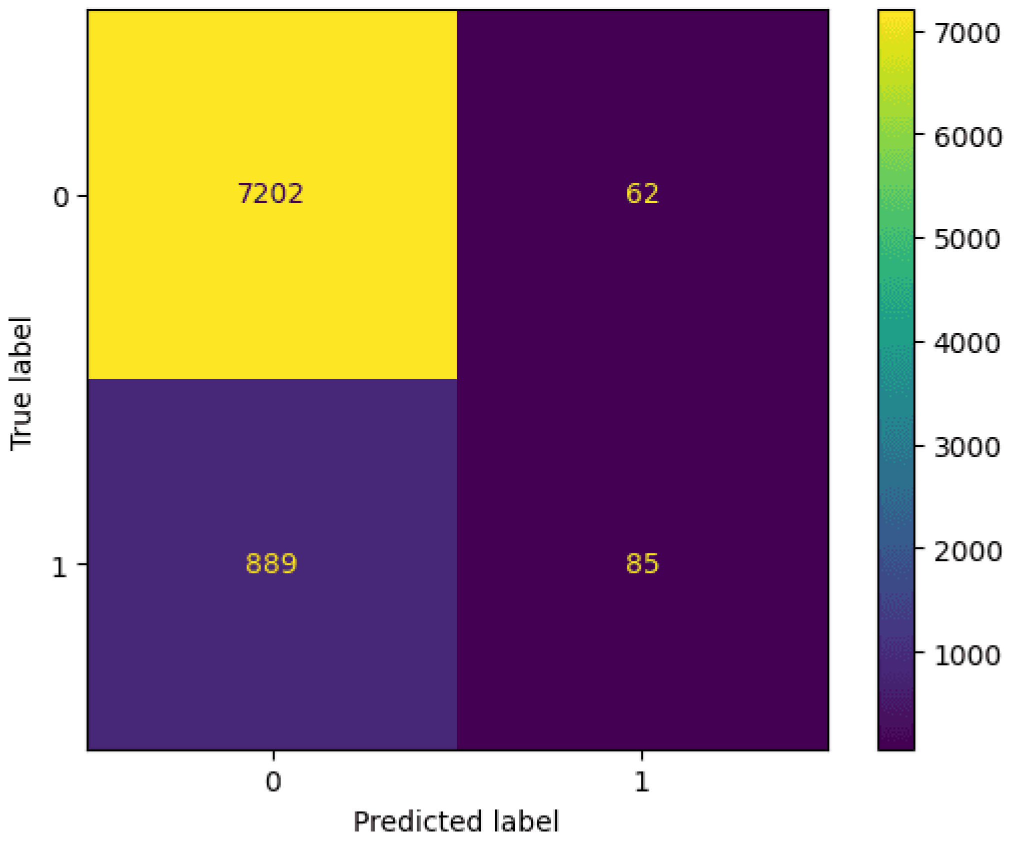
• The confusion matrix is a table that shows the number of true positives, true negatives, false positives, and false negatives for a classification model.

• Finally, the **ConfusionMatrixDisplay()** function is used to plot the confusion matrix.

• This function takes the confusion matrix as an argument and displays it as a heatmap.

• The resulting plot provides a visual representation of the model's performance in terms of correctly and incorrectly classified instances.

Output:



We should also evaluate the best model with accuracy, precision, and recall (note your results may differ due to randomization)

```
y_pred = knn.predict(X_test)


accuracy = accuracy_score(y_test, y_pred)

precision = precision_score(y_test, y_pred)

recall = recall_score(y_test, y_pred)


print("Accuracy:", accuracy)

print("Precision:", precision)
print("Recall:", recall)
```

Explanation:

This code snippet is evaluating the performance of a k-nearest neighbors (KNN) classifier on a test set.

• First, the **predict** method of the KNN classifier (**knn**) is used to generate predicted labels (**y_pred**) for the test set (**X_test**).

• Next, three performance metrics are calculated using the true labels (**y_test**) and the predicted labels (**y_pred**).

• The **accuracy_score** function from the **sklearn.metrics** module is used to calculate the accuracy of the classifier, which is the proportion of correctly classified instances.

• The **precision_score** function is used to calculate the precision of the classifier, which is the proportion of true positives (correctly classified positive instances) out of all instances classified as positive.

• The **recall_score** function is used to calculate the recall of the classifier, which is the proportion of true positives out of all actual positive instances.

• Finally, the accuracy, precision, and recall scores are printed to the console.

Output:

```
Accuracy: 0.885



Precision: 0.578
```

```
Recall: 0.0873
```

Explanation:

This code snippet is not actually code, but rather a set of performance metrics for a machine learning model.

• The first metric, "Accuracy," measures the proportion of correctly classified instances out of all instances.

• In this case, the model has an accuracy of 0.885, meaning it correctly classified 88.5% of instances.

• The second metric, "Precision," measures the proportion of true positives (correctly classified positive instances) out of all instances classified as positive.

• In this case, the model has a precision of 0.578, meaning it correctly classified 57.8% of positive instances.

• The third metric, "Recall," measures the proportion of true positives out of all actual positive instances.

• In this case, the model has a recall of 0.0873, meaning it correctly classified only 8.73% of actual positive instances.

• These metrics are commonly used to evaluate the performance of classification models.

The below code plots the importance of each feature, using the model's internal score to find the best way to split the data within each decision tree.

```
# Create a series containing feature importances from the model and feature names from the training data

feature_importances = pd.Series(best_rf.feature_importances_,
index=X_train.columns).sort_values(ascending=False)
```

```
# Plot a simple bar chart

feature_importances.plot.bar();
```

Explanation:

This code creates a bar chart of feature importances for a random forest model.

• First, a Pandas series is created called **feature_importances**.

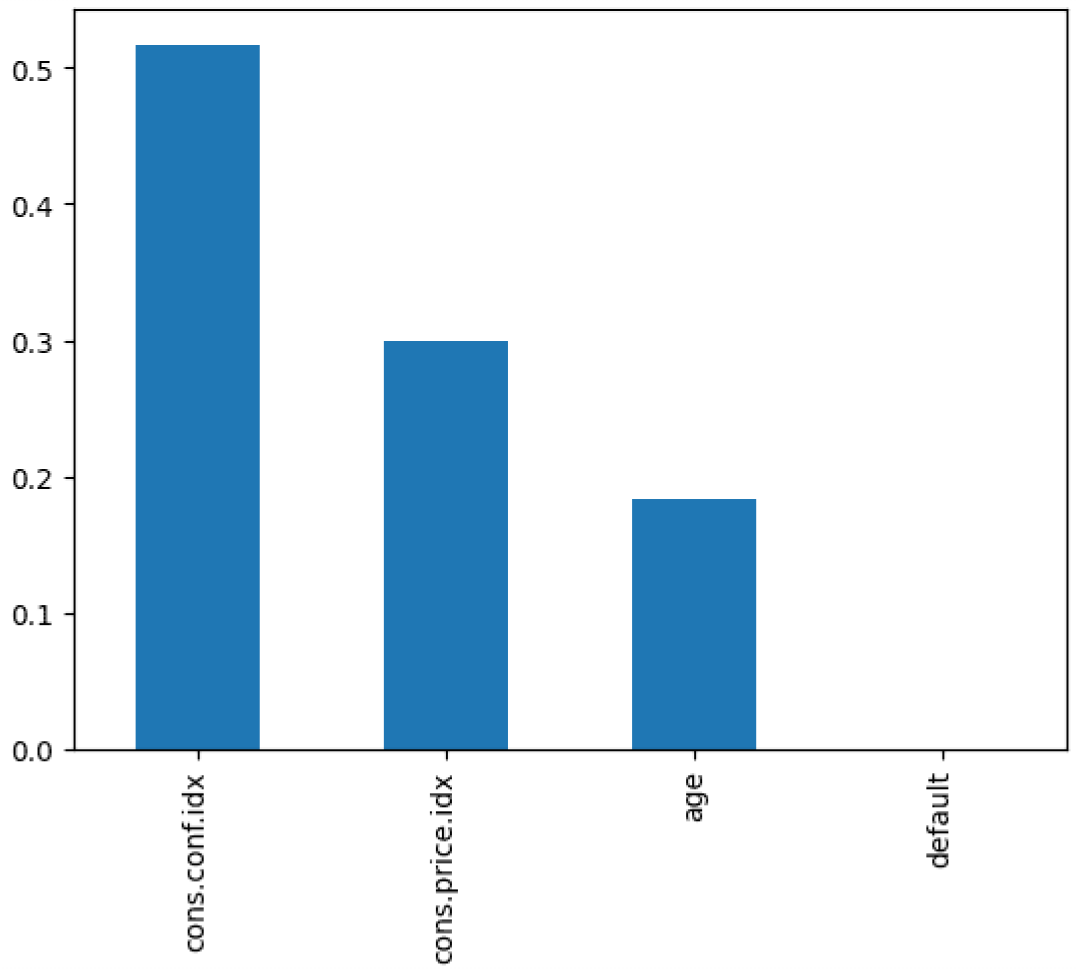
• This series contains the feature importances from the **best_rf** model (which is assumed to be a trained random forest model) and the feature names from the training data (**X_train.columns**).

• The **sort_values()** method is used to sort the feature importances in descending order.

• Next, the **plot.bar()** method is called on the **feature_importances** series to create a simple bar chart of the feature importances.

• Overall, this code is useful for visualizing the relative importance of different features in a random forest model.

This tells us that the consumer confidence index, at the time of the call, was the biggest predictor in whether the person subscribed.


0.5
0.4
0.3
0.2
0.1
0.0
cons.conf.idx
cons.price.idx
age
default