COSC 2753 | Machine Learning

Week 6 Lab Exercises: **Decision Trees**

Introduction

During the last couple of weeks we learned about the typical ML model development process. In this weeks lab we will explore decision tree based models.

The lab assumes that you have completed the labs for week 2-5. If you havent yet, please do so before attempting this lab.

The lab can be executed on either your own machine (with anaconda installation) or lab computer.

Objective

- Continue to familiarise with Python and other ML packages.
- Learning classification decision trees from both categorical and continuous numerical data
- Comparing the performance of various trees after pruned.
- Learning regression decision trees and comparing these models to regression models from previous labs.

Dataset

The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be (or not) subscribed.

Input variables:

- Bank client data:
    1. age (numeric)
    2. job : type of job (categorical: "admin.","unknown","unemployed","management","housemaid","entrepreneur","student", "blue-collar","self-employed","retired","technician","services")
    3. marital : marital status (categorical: "married","divorced","single"; note: "divorced" means divorced or widowed)
    4. education (categorical: "unknown","secondary","primary","tertiary")
    5. default: has credit in default? (binary: "yes","no")
    6. balance: average yearly balance, in euros (numeric)
    7. housing: has housing loan? (binary: "yes","no")
    8. loan: has personal loan? (binary: "yes","no")
- Related with the last contact of the current campaign:
    9. contact: contact communication type (categorical: "unknown","telephone","cellular")
    10. day: last contact day of the month (numeric)
    11. month: last contact month of year (categorical: "jan", "feb", "mar", ..., "nov", "dec")
    12. duration: last contact duration, in seconds (numeric)
- Other attributes:
    13. campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
    14. pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not previously contacted)
    15. previous: number of contacts performed before this campaign and for this client (numeric)
    16. poutcome: outcome of the previous marketing campaign (categorical: "unknown","other","failure","success")

Output variable (desired target):

```
17. y - has the client subscribed a term deposit? (binary: "yes","no")
```

This dataset is public available for research. The details are described in Moro et al., 2011.

Moro et al., 2011: S. Moro, R. Laureano and P. Cortez. Using Data Mining for Bank Direct Marketing: An Application of the CRISP-DM Methodology. In P. Novais et al. (Eds.), Proceedings of the European Simulation and Modelling Conference - ESM'2011, pp. 117-121, Guimarães, Portugal, October, 2011.

Lets read the data first.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 data = pd.read_csv('./Lab/bank-full.csv', delimiter=';')
6 data.head()
```

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	no

```
1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        45211 non-null  int64
 1   job        45211 non-null  object
 2   marital    45211 non-null  object
 3   education  45211 non-null  object
```

```
 4   default    45211 non-null  object
 5   balance    45211 non-null  int64
 6   housing    45211 non-null  object
 7   loan       45211 non-null  object
 8   contact    45211 non-null  object
 9   day        45211 non-null  int64
 10  month      45211 non-null  object
 11  duration   45211 non-null  int64
 12  campaign   45211 non-null  int64
 13  pdays      45211 non-null  int64
 14  previous   45211 non-null  int64
 15  poutcome   45211 non-null  object
 16  y          45211 non-null  object
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
```

The dataset contains categorical and numerical attributes. Lets convert the categorical columns to categorical data type in pandas.

```
for col in data.columns:
    if data[col].dtype == object:
        data[col] = data[col].astype('category')
```

sklearn's classification decision tree learner doesn't work with categorical attributes. It only works with continuous numeric attributes. The target class, however, must be categorical. So the categorical attributed must be converted into a suitable continuous format. Helpfully, Pandas can do this.

First, split the data into the target class and attributes:

```
dataY = data['y']
dataX = data.drop(columns='y')
```

Then use Pandas to generate "numerical" versions of the attributes:

```
dataXExpand = pd.get_dummies(dataX)
dataXExpand.head()
```

	age	balance	day	duration	campaign	pdays	previous	job_admin.	job_blue-collar	job_entrepreneur	...	month_jun	month_mar	month_may	month_nov	month_oct	month_sep	poutc
0	58	2143	5	261	1	-1	0	False	False	False	...	False	False	True	False	False	False	
1	44	29	5	151	1	-1	0	False	False	False	...	False	False	True	False	False	False	
2	33	2	5	76	1	-1	0	False	False	True	...	False	False	True	False	False	False	
3	47	1506	5	92	1	-1	0	False	True	False	...	False	False	True	False	False	False	
4	33	1	5	198	1	-1	0	False	False	False	...	False	False	True	False	False	False	

5 rows × 51 columns

```
dataXExpand.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 51 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   age                  45211 non-null  int64
 1   balance              45211 non-null  int64
 2   day                  45211 non-null  int64
 3   duration             45211 non-null  int64
 4   campaign             45211 non-null  int64
 5   pdays                45211 non-null  int64
 6   previous             45211 non-null  int64
 7   job_admin.           45211 non-null  bool
 8   job_blue-collar      45211 non-null  bool
 9   job_entrepreneur     45211 non-null  bool
 10  job_housemaid        45211 non-null  bool
 11  job_management       45211 non-null  bool
 12  job_retired          45211 non-null  bool
 13  job_self-employed    45211 non-null  bool
 14  job_services         45211 non-null  bool
 15  job_student          45211 non-null  bool
 16  job_technician       45211 non-null  bool
 17  job_unemployed       45211 non-null  bool
 18  job_unknown          45211 non-null  bool
 19  marital_divorced     45211 non-null  bool
 20  marital_married      45211 non-null  bool
 21  marital_single       45211 non-null  bool
 22  education_primary    45211 non-null  bool
 23  education_secondary  45211 non-null  bool
 24  education_tertiary   45211 non-null  bool
 25  education_unknown    45211 non-null  bool
 26  default_no           45211 non-null  bool
 27  default_yes          45211 non-null  bool
 28  housing_no           45211 non-null  bool
 29  housing_yes          45211 non-null  bool
 30  loan_no              45211 non-null  bool
 31  loan_yes             45211 non-null  bool
 32  contact_cellular     45211 non-null  bool
 33  contact_telephone    45211 non-null  bool
 34  contact_unknown      45211 non-null  bool
 35  month_apr            45211 non-null  bool
 36  month_aug            45211 non-null  bool
 37  month_dec            45211 non-null  bool
 38  month_feb            45211 non-null  bool
 39  month_jan            45211 non-null  bool
 40  month_jul            45211 non-null  bool
 41  month_jun            45211 non-null  bool
 42  month_mar            45211 non-null  bool
 43  month_may            45211 non-null  bool
 44  month_nov            45211 non-null  bool
 45  month_oct            45211 non-null  bool
 46  month_sep            45211 non-null  bool
 47  poutcome_failure     45211 non-null  bool
 48  poutcome_other       45211 non-null  bool
 49  poutcome_success     45211 non-null  bool
 50  poutcome_unknown     45211 non-null  bool
dtypes: bool(44), int64(7)
memory usage: 4.3 MB
```

As you can see, the categories are expanded into boolean (yes/no, that is, 1/0) values that can be treated as continuous numerical values. It’s not ideal, but it will allow a correct decision tree to be learned.

🔶 Why is it necessary to convert the attributes into boolean representations, rather than just convert them into integer values? What problem would be caused by converting the attributes into integers?

1. **Avoiding Implicit Ordering**: When you convert categorical data into integers, it introduces an artificial order or hierarchy that doesn't exist in the original data. For example, if you have a 'job' column with categories like 'admin', 'technician', 'entrepreneur', and you map them to integers like 1, 2, 3, the algorithm might incorrectly interpret these numbers as having an order (i.e., 'technician' > 'admin', 'entrepreneur' > 'technician'), which can lead to skewed or incorrect results.
2. **Equal Representation**: One-hot encoding treats all categories equally without implying any sort of ordinal relationship between them. Each category becomes a separate feature with equal weight, which is more representative of the true nature of categorical data.
3. **Model Compatibility**: Many machine learning models are designed to work with continuous data and can misinterpret the nature of the data if it's simply converted to integers. One-hot encoding transforms categorical data into a binary matrix that these models can process more effectively.

Problems with Converting Attributes into Integers:

1. **Incorrect Assumptions by the Model**: As mentioned, the model might infer a non-existent ordinal relationship between categories. This can lead the model to draw false conclusions about the data.
2. **Distorted Distance Measurements**: In models that calculate distances between data points (like in K-Nearest Neighbors), integer encoding can cause distortion. The model might assume that two data points are close to each other because their integer representations are numerically close, even if they're actually quite different.
3. **Limiting Model Performance**: Integer encoding can limit the ability of the model to learn complex patterns in the data, especially if the categorical variable has no inherent order. This can result in poorer model performance.

The target class also needs to be pre-processed. The target will be treated by sklearn as a category, but sklearn requires that these categories are represented as integers (not strings). To convert the strings into numbers, the preprocessing. LabelEncoder class from sklearn can be used, as shown below. The two print statements show how to convert in both directions (strings to integers, and vice-versa).

```
from sklearn import preprocessing

le = preprocessing.LabelEncoder()
le.fit(dataY)
class_labels = le.inverse_transforam([0,1])
dataY = le.transform(dataY)
```

1. `le = preprocessing.LabelEncoder()` : This line is creating an instance of the `LabelEncoder` class.
2. `le.fit(dataY)` : The `fit` method is used to fit the label encoder instance to the data. This means it examines all the unique values in `dataY`, and assigns each unique value to a unique integer, which will be used to transform the categorical data into numerical data. The unique values are stored in the `classes_` attribute of the `LabelEncoder` instance.
3. `class_labels = le.inverse_transform([0,1])` : The `inverse_transform` method is used to convert the encoded numerical values back into their original categorical form. In this case, it's being used to get the original categorical values for the encoded values 0 and 1. These original values are stored in the `class_labels` variable.
4. `dataY = le.transform(dataY)` : The `transform` method is used to convert the categorical data in `dataY` into numerical data. It does this by replacing each unique value in `dataY` with the integer that was assigned to it when the `fit` method was called. The transformed data is then stored back into the `dataY` variable

We already convert "no"/"yes" into [0,1], try to print labels for checking

```
print(dataY)
print(len(dataY))
print(class_labels)
```

```
[0 0 0 ... 1 0 0]
45211
['no' 'yes']
```

EDA

> **☞ Task: Since we have covered how to do EDA in the previous labs, this section is left as an exercise for you. Complete the EDA and use the information to justify the decisions made in the subsequent code blocks.**

[ ] ↳ *4 cells hidden*

Setting up the performance (evaluation) metric

There are many performance metrics that apply to this problem such as `accuracy_score`, `f1_score`, etc. More information on performance metrics available in sklearn can be found at: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

The insights gained in the EDA becomes vital in determining the performance metric. Try to identify the characteristics that are important in making this decision from the EDA results. Use your judgment to pick the best performance measure - discuss with the lab demonstrator to see if the performance measure you came up with is appropriate.

In this task, I want to give equal importance to all classes. Therefore I will select `macro–averaged f1_score` as my performance measure and I wish to achieve a target value of 75% f1_score.

F1-score is NOT the only performance measure that can be used for this problem.

Setup the experiment - data splits

Next **what data should we use to evaluate the performance?**

We can generate "simulated" unseen data in several methods

1. Hold-Out validation
2. Cross-Validation

Lets use hold out validation for this experiment.

> ☞ **Task: Use the knowledge from last couple of weeks to split the data appropriately.**

```
from sklearn.model_selection import train_test_split

#with pd.option_context('mode.chained_assignment', None):
train_data_X_, test_data_X, train_data_y_ , test_data_y = train_test_split(dataXExpand, dataY, test_size=0.2,
                                            shuffle=True,random_state=0)

#with pd.option_context('mode.chained_assignment', None):
train_data_X, val_data_X, train_data_y, val_data_y = train_test_split(train_data_X_, train_data_y_, test_size=0.25,
                                          shuffle=True,random_state=0)

print(train_data_X.shape, val_data_X.shape, test_data_X.shape)
```

```
(27126, 51) (9042, 51) (9043, 51)
```

```
train_X = train_data_X.to_numpy()
train_y = train_data_y

test_X = test_data_X.to_numpy()
test_y = test_data_y

val_X = val_data_X.to_numpy()
val_y = val_data_y
```

Lets setup few functions to visualise the results.

It is likely that you won't have the graphviz package available, in which case you will need to install graphviz. This can be done through the anacoda-navigator interface (environment tab):

1. Change the dropbox to "All"
2. Search for the packagec python-graphviz
3. Select the python-graphviz package and install (press "apply")

If you cant install graphviz don't worry - you can still complete the lab. Graphviz is nice to be able to see the trees that are being calculated. However, once the trees become complex, visualising them isn't practical. More can be found here: https://anaconda.org/conda-forge/python-graphviz

```
import graphviz

def get_tree_2_plot(clf):
    dot_data = tree.export_graphviz(clf, out_file=None,
                      feature_names=dataXExpand.columns,
                      class_names=class_labels,
                      filled=True, rounded=True,
                      special_characters=True)
    graph = graphviz.Source(dot_data)
    return graph
```

The provided Python code defines a function named `get_tree_2_plot` that takes a decision tree classifier (`clf`) as an argument and returns a Graphviz representation of the decision tree.

Here's a step-by-step explanation of what the function does:

1. The function starts by calling the `export_graphviz` function from the `tree` module of the `sklearn` library. This function generates a Graphviz representation of the decision tree, which is a type of diagram used to represent hierarchical data structures. The `export_graphviz` function takes several arguments:
    - `clf`: This is the decision tree classifier that you want to visualize.
    - `out_file`: This argument is set to `None`, which means the function will return the Graphviz representation as a string instead of writing it to a file.
    - `feature_names`: This argument is set to `dataXExpand.columns`, which should be a list of the names of the features used in the decision tree.
    - `class_names`: This argument is set to `class_labels`, which should be a list of the names of the classes that the decision tree can predict.
    - `filled`: When this argument is set to `True`, the nodes of the decision tree will be colored.
    - `rounded`: When this argument is set to `True`, the boxes representing the nodes of the decision tree will have rounded corners.
    - `special_characters`: When this argument is set to `True`, special characters will be allowed in the Graphviz representation.
2. The Graphviz representation generated by the `export_graphviz` function is stored in the `dot_data` variable.
3. The `dot_data` string is then passed to the `Source` function from the `graphviz` library, which creates a Graphviz object that can be displayed in Jupyter notebooks or saved to a file.
4. The Graphviz object is returned by the function, allowing it to be used elsewhere in your code.

In summary, this function takes a decision tree classifier and returns a visual representation of the decision tree that can be displayed or saved to a file.

```
from sklearn.metrics import f1_score

def get_acc_scores(clf, train_X, train_y, val_X, val_y):
    train_pred = clf.predict(train_X)
    val_pred = clf.predict(val_X)

    train_acc = f1_score(train_y, train_pred, average='macro')
    val_acc = f1_score(val_y, val_pred, average='macro')

    return train_acc, val_acc
```

Simple decision tree training

Lets train a simple decision tree and visualize it.

```
from sklearn import tree

tree_max_depth = 2   #change this value and observe

clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=tree_max_depth, class_weight='balanced')
clf = clf.fit(train_X, train_y)
```

```
Dtree = get_tree_2_plot(clf)
Dtree
```

Here's a step-by-step explanation of what the code does:

1. `tree_max_depth = 2`: This line is setting the maximum depth of the decision tree to 2. The depth of a tree is the maximum distance between the root and any leaf. A smaller depth can help to prevent overfitting, which is when the model learns the training data too well and performs poorly on new, unseen data. However, if the depth is too small, the model may underfit the data, which means it may not learn enough from the training data to make accurate predictions.
2. `clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=tree_max_depth, class_weight='balanced')`: This line is creating an instance of the `DecisionTreeClassifier` class. The `criterion` parameter is set to `'entropy'`, which means the decision tree will use the entropy impurity measure to decide where to split the data. The `max_depth` parameter is set to the value of `tree_max_depth`, which is 2. The `class_weight` parameter is set to `'balanced'`, which means the algorithm will adjust the weights of the classes to be inversely proportional to their frequency, which can help when dealing with imbalanced datasets.
3. `clf = clf.fit(train_X, train_y)`: The `fit` method is used to train the decision tree classifier on the training data. `train_X` should be a 2D array-like object where each row is a different sample and each column is a different feature. `train_y` should be a 1D array-like object where each element is the class label of the corresponding sample in `train_X`.

In summary, this code is creating and training a decision tree classifier with a maximum depth of 2, using the entropy impurity measure and balanced class weights.

```
train_acc, val_acc = get_acc_scores(clf,train_X, train_y, val_X, val_y)
print("Train f1 score: {:.3f}".format(train_acc))
print("Validation f1 score: {:.3f}".format(val_acc))
```

```
Train f1 score: 0.552
Validation f1 score: 0.560
```

 Did we achieve the desired target value? If not what do you thing the above results indicate: over-fitting, under-fitting

 Based on the answer to the above question, what do you think is the best course of action?

Based on the F1 scores provided:

- Train F1 score: 0.552
- Validation F1 score: 0.560

It seems that the target F1 score of 0.75 was not achieved. Instead, the model produced a macro-averaged F1 score around 0.55 on both the training and validation sets. Here's what these results may indicate:

- **Underfitting**: The similar performance on both training and validation datasets, with a low score, suggests that the model may be underfitting. Underfitting occurs when a model is too simple to capture the underlying pattern in the data. This could be due to overly simplistic model choice, not enough features, or not capturing the relationships between features and the target variable effectively.
- **Complexity of Model**: If the decision tree is too shallow (not deep enough), it may not have the capacity to learn the nuances of the data. Considering increasing the depth of the tree or relaxing some of the constraints that might be preventing it from growing more complex decision boundaries.
- **Quality of Features**: The features used may not be predictive enough of the target variable, or important features might be missing. Feature engineering or selection might improve the model's performance.
- **Data Imbalance**: If the classes in the target variable are imbalanced, this might skew the F1 score lower. In such cases, you might want to look into methods to handle imbalanced data, such as resampling techniques or different decision thresholds.
- **Noise in Data**: There may be noise in the data that's making it hard for the model to learn. Ensuring data quality and preprocessing to remove noise can sometimes improve model performance.
- **Algorithm Choice**: Decision trees are a good starting point, but they might not always be the best model choice for every problem. You could consider trying other algorithms like Random Forest, Gradient Boosting, or even neural networks, which might be able to capture more complex patterns.

Considering the potential for underfitting and the current F1 scores, here are some steps you could take to improve your model:

1. **Model Complexity**:
   - Evaluate the complexity of your current decision tree. If it's too simple, it may not capture the patterns properly. Try increasing the maximum depth (`max_depth`) of the tree.
   - Relax other constraints like `min_samples_split`, `min_samples_leaf`, or `max_leaf_nodes`.
2. **Feature Engineering**:
   - Perform feature selection to identify the most important features that contribute to the target variable.
   - Create new features that might better capture the nuances of the data.
   - Normalize or scale features if you haven't already, as some models can be sensitive to the scale of the data.
3. **Data Quality and Relevance**:
   - Revisit the data cleaning and preprocessing steps to ensure that noise is minimized and that the data is as relevant and clean as possible.
   - Handle missing or outlier values appropriately if not already done.
4. **Handling Imbalanced Data**:
   - If the dataset is imbalanced, consider using techniques like SMOTE (Synthetic Minority Over-sampling Technique) for oversampling the minority class or under-sampling the majority class.
   - Adjust the decision threshold to cater to the minority class or use class weights if supported by the classifier.
5. **Ensemble Methods**:

- Use ensemble learning methods such as Random Forests or Gradient Boosting, which can often outperform a single decision tree and are less prone to overfitting.
- Ensemble models also have the advantage of capturing more complex patterns in the data.

6. **Cross-Validation**:
    - Perform k-fold cross-validation to ensure that the model's performance is consistent across different subsets of the data.
    - Cross-validation helps in identifying if the model has stability regarding its performance.

Hyper parameter tuning

🔍 What are the hyper parameters of the `DecisionTreeClassifier`?

You may decide to tune the important hyper-paramters of the decision tree classifier (identified in the above question) to get the best performance. As an example I have selected two hyper parameters: `max_depth` and `min_samples_split`.

In this exercise I will be using GridSearch to tune my parameters. Sklearn has a function that do cross validation to tune the hyper parameters called `GridSearchCV`. Lets use this function.

This step may take several steps depending on the performance of your computer

The `DecisionTreeClassifier` from scikit-learn has several hyperparameters that control the learning process. Adjusting these hyperparameters can greatly affect the performance of the model. Here are some of the key hyperparameters:

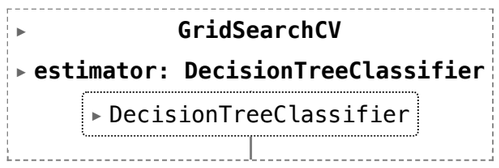
1. **`criterion`**: The function used to measure the quality of a split. Supported criteria are `'gini'` for the Gini impurity and `'entropy'` for the information gain.
2. **`splitter`**: The strategy used to choose the split at each node. Supported strategies are `'best'` to choose the best split and `'random'` to choose the best random split.
3. **`max_depth`**: The maximum depth of the tree. If `None`, then nodes are expanded until all leaves contain less than `min_samples_split` samples.
4. **`min_samples_split`**: The minimum number of samples required to split an internal node. It can be an integer (the minimum number) or a float (a fraction of the total number of samples).
5. **`min_samples_leaf`**: The minimum number of samples required to be at a leaf node. Similar to `min_samples_split`, it can be an integer or a float.
6. **`min_weight_fraction_leaf`**: The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node.
7. **`max_features`**: The number of features to consider when looking for the best split. It can be an integer, float, string (`'auto'`, `'sqrt'`, `'log2'`), or `None`.
8. **`random_state`**: Controls the randomness of the estimator. The features are always randomly permuted at each split, even if `splitter` is set to `'best'`.
9. **`max_leaf_nodes`**: The maximum number of leaf nodes a tree can have. If `None`, then unlimited number of leaf nodes are allowed.
10. **`min_impurity_decrease`**: A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
11. **`class_weight`**: Weights associated with classes. If not given, all classes are supposed to have weight one. For imbalanced datasets, it can be set to `'balanced'`.
12. **`ccp_alpha`**: Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen.

These hyperparameters allow you to control the growth and complexity of the decision tree, helping to prevent overfitting (by restricting the tree size) or underfitting (by allowing the tree to be more complex). The process of choosing the best hyperparameters usually involves experimenting with different combinations and using techniques like grid search or randomized search with cross-validation.

```
from sklearn.model_selection import GridSearchCV

parameters = {'max_depth':np.arange(2,400, 50), 'min_samples_split':np.arange(2,50,5)}

dt_clf = tree.DecisionTreeClassifier(criterion='entropy', class_weight='balanced')
Gridclf = GridSearchCV(dt_clf, parameters, scoring='f1_macro')
Gridclf.fit(train_X, train_y)
```



Here's a step-by-step explanation of what the code does:

1. `parameters = {'max_depth':np.arange(2,400, 50), 'min_samples_split':np.arange(2,50,5)}`: This line is defining the grid of hyperparameters to search over. The `max_depth` parameter, which controls the maximum depth of the tree, will be tested with values from 2 to 400 in steps of 50. The `min_samples_split` parameter, which controls the minimum number of samples required to split an internal node, will be tested with values from 2 to 50 in steps of 5.
2. `dt_clf = tree.DecisionTreeClassifier(criterion='entropy', class_weight='balanced')`: This line is creating an instance of the `DecisionTreeClassifier` class. The `criterion` parameter is set to `'entropy'`, which means the decision tree will use the entropy impurity measure to decide where to split the data. The `class_weight` parameter is set to `'balanced'`, which means the algorithm will adjust the weights of the classes to be inversely proportional to their frequency, which can help when dealing with imbalanced datasets.
3. `Gridclf = GridSearchCV(dt_clf, parameters, scoring='f1_macro')`: This line is creating an instance of the `GridSearchCV` class, which will perform the grid search. The `GridSearchCV` instance is initialized with the decision tree classifier, the grid of parameters to search over, and the scoring method to use. The `scoring` parameter is set to `'f1_macro'`, which means the F1 score will be calculated separately for each class and then averaged (not taking the class imbalance into account).
4. `Gridclf.fit(train_X, train_y)`: The `fit` method is used to perform the grid search. The method will train a decision tree classifier for each combination of parameters in the grid, evaluate the performance of each classifier using cross-validation, and then select the parameters that resulted in the best performance.

In summary, this code is performing a grid search to find the best hyperparameters for a decision tree classifier, using the F1 score as the performance metric.

```
pd.DataFrame(Gridclf.cv_results_)
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_min_samples_split	params	split0_test_score	split1_test_score	split2_test
0	0.131600	0.010323	0.021427	0.003447	2	2	{'max_depth': 2, 'min_samples_split': 2}	0.555917	0.523732	0
1	0.162010	0.069471	0.020250	0.002544	2	7	{'max_depth': 2, 'min_samples_split': 7}	0.555917	0.523732	0
2	0.125370	0.005250	0.020743	0.003202	2	12	{'max_depth': 2, 'min_samples_split': 12}	0.555917	0.523732	0
3	0.123643	0.003293	0.018379	0.001947	2	17	{'max_depth': 2, 'min_samples_split': 17}	0.555917	0.523732	0
4	0.122846	0.001760	0.020015	0.002542	2	22	{'max_depth': 2, 'min_samples_split': 22}	0.555917	0.523732	0
...	...	...	...	...	...	...	...	...	...	
75	0.454474	0.036304	0.020027	0.004339	352	27	{'max_depth': 352, 'min_samples_split': 27}	0.714956	0.696693	0
76	0.428243	0.012163	0.023232	0.004137	352	32	{'max_depth': 352, 'min_samples_split': 32}	0.717190	0.699079	0
77	0.424863	0.008867	0.020299	0.000462	352	37	{'max_depth': 352, 'min_samples_split': 37}	0.709173	0.696130	0
78	0.413947	0.011447	0.018691	0.002036	352	42	{'max_depth': 352, 'min_samples_split': 42}	0.711864	0.697751	0
79	0.413725	0.021600	0.020919	0.001494	352	47	{'max_depth': 352, 'min_samples_split': 47}	0.712011	0.703407	0

80 rows × 15 columns

```
print(Gridclf.best_score_)
print(Gridclf.best_params_)

clf = Gridclf.best_estimator_
```

```
0.7113728672093905
{'max_depth': 202, 'min_samples_split': 47}
```

```
train_acc, val_acc = get_acc_scores(clf,train_X, train_y, val_X, val_y)
print("Train f1 score: {:.3f}".format(train_acc))
print("Validation f1 score: {:.3f}".format(val_acc))
```

```
Train f1 score: 0.784
Validation f1 score: 0.725
```

- Did we achieve the desired target value? If not what do you thing the above results indicate: over-fitting, under-fitting
- Based on the answer to the above question, what do you think is the best course of action?

Based on the results provided:

- **Best cross-validation F1 score**: 0.711
- **Training F1 score**: 0.784
- **Validation F1 score**: 0.725

These results suggest that the desired target F1 score of 0.75 has not been achieved on the validation set, although it's very close. The training F1 score is higher than the validation F1 score, which often indicates a tendency towards overfitting. Overfitting occurs when the model learns patterns that are specific to the training data, which do not generalize well to unseen data.

However, it's worth noting that the gap between the training and validation F1 scores isn't very large, which suggests that while the model may be slightly overfitting, it's not severe.

Best Course of Action:

1. **Evaluate on Test Set**: Before making any further decisions, evaluate the model on the test set to see if the validation F1 score is reflective of the model's performance on unseen data.
2. **Tune Complexity**: If there's overfitting, consider reducing the complexity of the model slightly. This could mean reducing the `max_depth` or increasing the `min_samples_split` to prevent the model from learning overly specific patterns.
3. **Additional Hyperparameter Tuning**: Further refine other hyperparameters around the current best values, perhaps in smaller increments, to see if a better balance between bias and variance can be achieved.
4. **Feature Engineering**: If you have not exhausted feature engineering, consider creating new features or transforming existing ones. Feature importance analysis can guide you on which features to focus on.
5. **Ensemble Methods**: Since decision trees are the base learners for ensemble methods like Random Forests or Gradient Boosting, using these methods might improve performance without overfitting. They work by building multiple trees and aggregating their predictions, which often leads to better generalization.
6. **Regularization Techniques**: Explore the use of additional regularization techniques, such as setting `min_impurity_decrease` or using the `ccp_alpha` parameter for cost complexity pruning, to simplify the model and potentially improve validation and test scores.
7. **Cross-Validation**: Ensure that the model selection process is robust by using k-fold cross-validation, which reduces the variance of the model's performance estimate.
8. **Model Interpretability**: If the decision tree's complexity is making it hard to interpret, consider whether a simpler model might suffice. Sometimes a small decrease in performance is acceptable in exchange for a model that is easier to understand and explain.
9. **Problem Complexity**: Consider whether the F1 score of 0.75 is a realistic target for this problem given the features and data quality. It might be that the data does not contain enough signal to reach this level of performance.

Post pruning decision trees with cost complexity pruning

The DecisionTreeClassifier provides parameters such as min_samples_leaf and max_depth to prevent a tree from overfitting. Those parameters prevent the tree from growing to large size and are examples of pre pruning.

Cost complexity pruning provides another option to control the size of a tree. In DecisionTreeClassifier, this pruning technique is parameterized by the cost complexity parameter, ccp_alpha. Greater values of ccp_alpha increase the number of nodes pruned. Here we only show the effect of ccp_alpha on regularizing the trees and how to choose a ccp_alpha based on validation scores.

Minimal cost complexity pruning recursively finds the node with the “weakest link”. The weakest link is characterized by an effective alpha, where the nodes with the smallest effective alpha are pruned first. To get an idea of what values of ccp_alpha could be appropriate, scikit-learn provides DecisionTreeClassifier.cost_complexity_pruning_path that returns the effective alphas and the corresponding total leaf impurities at each step of the pruning process. As alpha increases, more of the tree is pruned, which increases the total impurity of its leaves.

Note that min_samples_leaf and max_depth are pre-pruning techniques, while the cost complexity pruning algorithm works after the tree is grown and this is a post pruning technique.

You can check out more detail here: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html

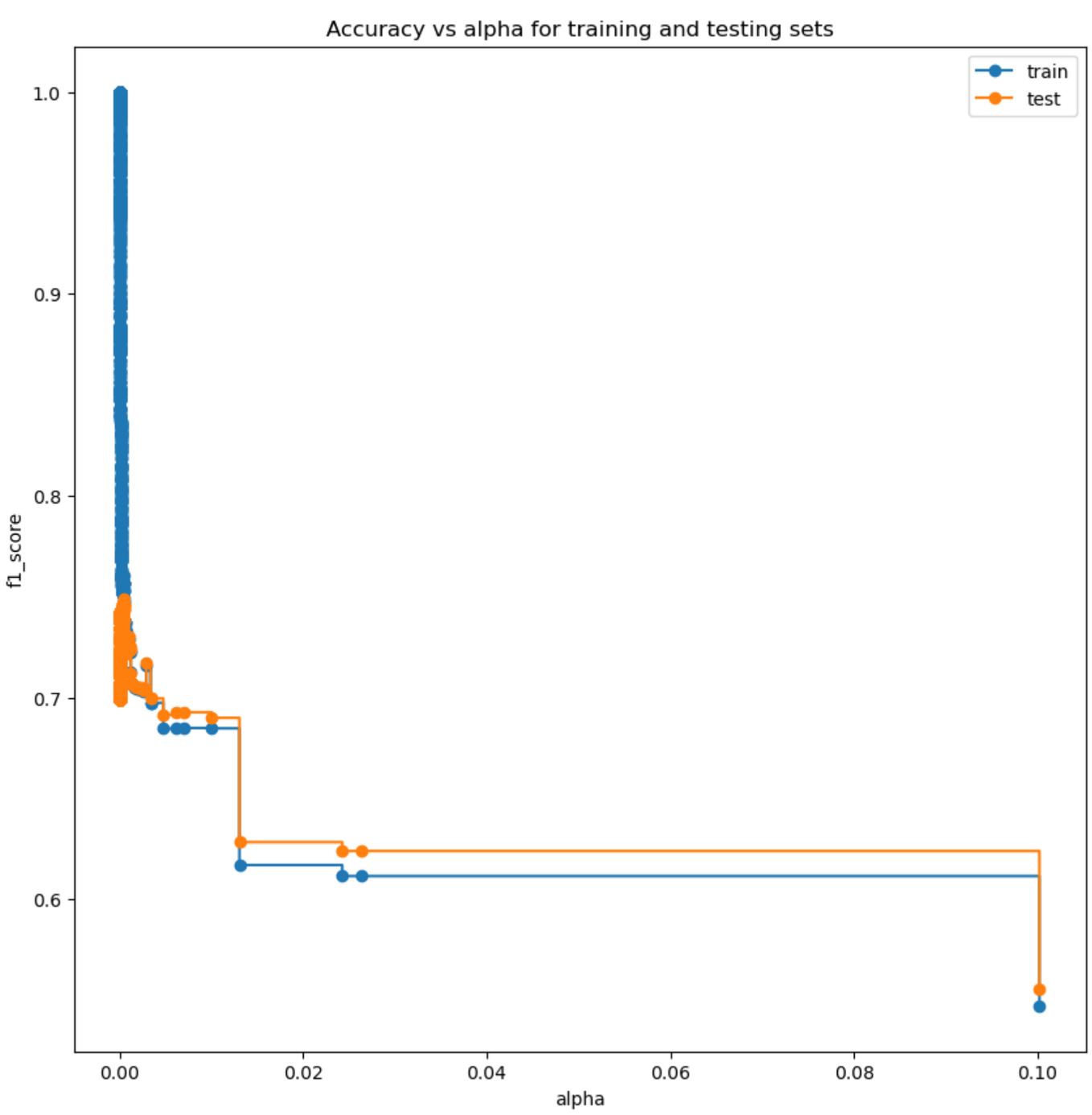
```
clf = tree.DecisionTreeClassifier(class_weight='balanced')
path = clf.cost_complexity_pruning_path(train_X, train_y)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

The following step may take several steps depending on the performance of your computer

```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = tree.DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha, class_weight='balanced')
    clf.fit(train_X, train_y)
    clfs.append(clf)
```

```
train_scores = [f1_score(train_y, clf.predict(train_X), average='macro') for clf in clfs]
val_scores = [f1_score(val_y, clf.predict(val_X), average='macro') for clf in clfs]

fig, ax = plt.subplots(figsize=(10,10))
ax.set_xlabel("alpha")
ax.set_ylabel("f1_score")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker='o', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas, val_scores, marker='o', label="test",
        drawstyle="steps-post")
ax.legend()
plt.show()
```

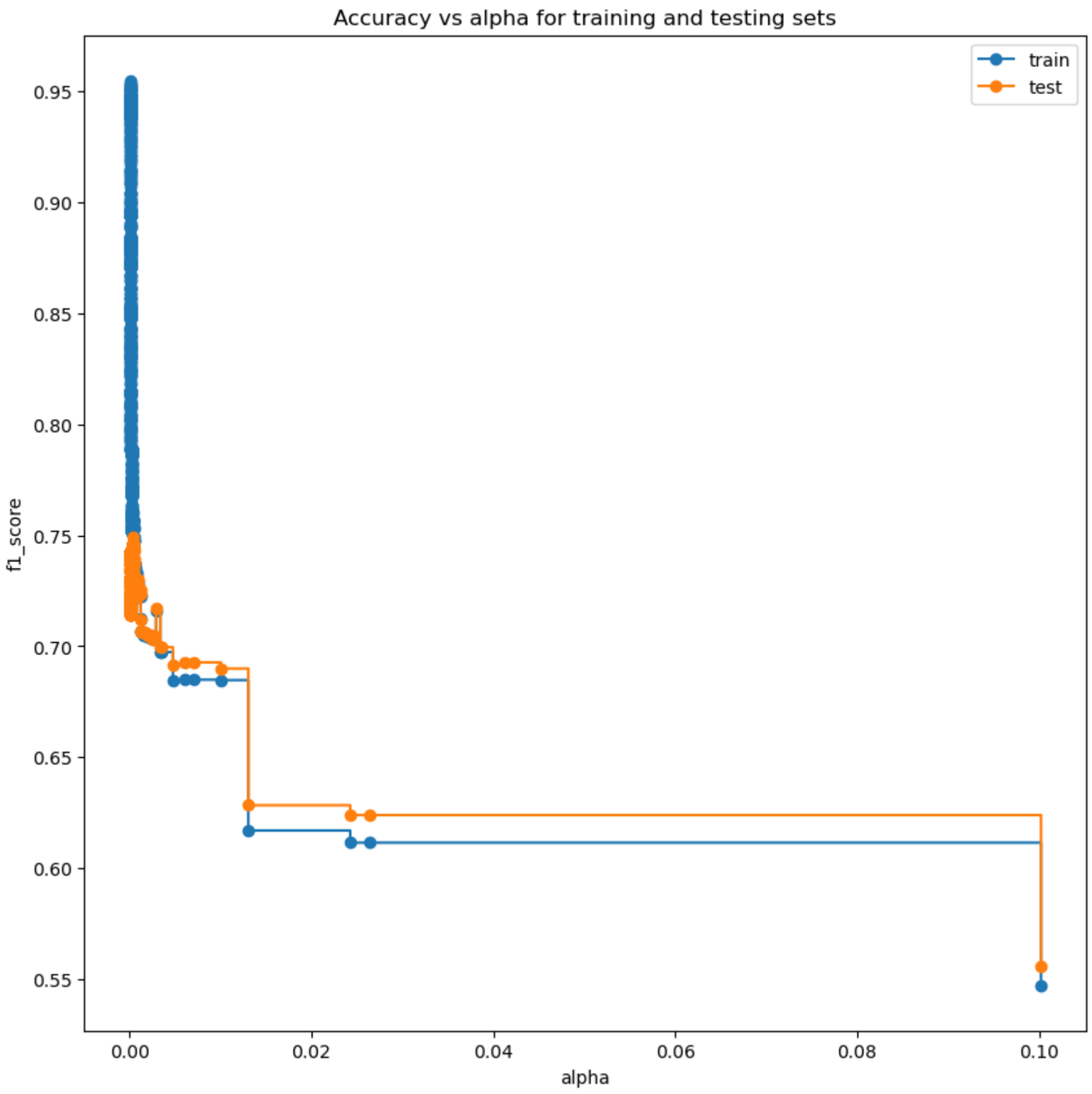


What `ccp_alphas` value would you choose as the best for the task?

Look more detail to see!

```
#train_scores = [f1_score(train_y, clf.predict(train_X), average='macro') for clf in clfs]
#val_scores = [f1_score(val_y, clf.predict(val_X), average='macro') for clf in clfs]


fig, ax = plt.subplots(figsize=(10,10))
ax.set_xlabel("alpha")
ax.set_ylabel("f1_score")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas[700:1500], train_scores[700:1500], marker='o', label="train",
        drawstyle="steps-post")
ax.plot(ccp_alphas[700:1500], val_scores[700:1500], marker='o', label="test",
        drawstyle="steps-post")
ax.legend()
plt.show()
```




```
# Example Python code to find the optimal ccp_alpha value
optimal_index = val_scores.index(max(val_scores))
optimal_ccp_alpha = ccp_alphas[optimal_index]
print(optimal_ccp_alpha)

```

```
0.00043235981997801366
```

Choosing the best `ccp_alpha` value involves finding a balance between the model's ability to generalize to unseen data (validation set) and its performance on the training set. The goal is to select a value that maximizes the F1 score on the validation set without compromising too much on the training set performance.

Based on the second, zoomed-in graph provided, we're looking for the peak of the validation (test in the legend) F1 score curve. The exact `ccp_alpha` value at this peak isn't explicitly provided in the output, but we can observe from the graph where the validation F1 score is highest before it starts to decline.

It appears that the peak occurs at an `alpha` value shortly before the steep drop in F1 score for the validation set. This point represents the best trade-off between bias and variance, where increasing the pruning strength any further would lead to underfitting, and any less would not sufficiently regularize the model, potentially leading to overfitting.

If the x-axis of your graph is linear and evenly spaced, you can estimate the `ccp_alpha` value visually. If the training and validation scores are plotted at regular intervals of `ccp_alpha`, you might be able to identify the approximate index of the highest validation score and then map that back to the corresponding `ccp_alpha` value from your array of `ccp_alphas`.

In practice, you would programmatically find this value by examining the `val_scores` list and finding the index of the maximum F1 score, then retrieving the corresponding `ccp_alpha`:

```
# Example Python code to find the optimal ccp_alpha value
optimal_index = val_scores.index(max(val_scores))
optimal_ccp_alpha = ccp_alphas[optimal_index]
```

This `optimal_ccp_alpha` would be your chosen value for the task.

Remember, before making a final decision, it is important to verify this `ccp_alpha` value by evaluating the decision tree's performance on a separate test set that was not used during the training or validation process. If this separate test set F1 score is also satisfactory, it further validates your choice of `ccp_alpha`.

Random forest

Lets make many trees using our dataset. If we run the DT algorithm multiple times on same data, it will result in the same tree. To make different trees we can inject some randomness. Select data data points and features to be used in DT algorithm randomly - this process is called creating boot strapped datasets.

This is automatically done in sklearn for us in the `RandomForestClassifier`. Lets use that on our dataset.

More code can be found here: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=8, n_estimators=500, class_weight='balanced_subsample', random_state=0)
clf.fit(train_X, train_y)
```

```
RandomForestClassifier(class_weight='balanced_subsample', max_depth=8,
                       n_estimators=500, random_state=0)
```

```
train_acc, val_acc = get_acc_scores(clf, train_X, train_y, val_X, val_y)
print("Train Accuracy score: {:.3f}".format(train_acc))
print("Validation Accuracy score: {:.3f}".format(val_acc))
```

```
Train Accuracy score: 0.745
Validation Accuracy score: 0.731
```

> ☞ **Task: Now tune the hyper parameters of the random forest.**

```
clf = RandomForestClassifier(max_depth=5, n_estimators=500, class_weight='balanced_subsample', random_state=0)
clf.fit(train_X, train_y)
```

```
RandomForestClassifier(class_weight='balanced_subsample', max_depth=5,
                       n_estimators=500, random_state=0)
```

```
train_acc, val_acc = get_acc_scores(clf, train_X, train_y, val_X, val_y)
print("Train Accuracy score: {:.3f}".format(train_acc))
print("Validation Accuracy score: {:.3f}".format(val_acc))
```

```
Train Accuracy score: 0.707
Validation Accuracy score: 0.711
```

 Is the final model that you get after hyper parameter tuning better than the previous decision tree model? Why?

Lets visualise the feature importance of the RF classifier.

The final model, a RandomForestClassifier with `max_depth=5`, `n_estimators=500`, and using `balanced_subsample` for class weighting, achieved:

- **Train F1 score**: 0.707
- **Validation F1 score**: 0.711

To evaluate whether this model is better than the previous decision tree model, we need to compare these scores with the decision tree's performance. From your earlier messages, the best decision tree model had:

- **Train F1 score**: 0.784
- **Validation F1 score**: 0.725

Here's how to interpret the comparison:

- **Performance on Training Data**: The decision tree has a higher F1 score on the training data compared to the random forest. This might indicate that the decision tree is capturing the relationships in the training data more thoroughly, potentially overfitting.
- **Performance on Validation Data**: The decision tree also has a higher F1 score on the validation set. This suggests that, despite the higher complexity of the decision tree (as indicated by the higher training F1 score), it is still generalizing better than the random forest with these particular hyperparameters.
- **Complexity and Variance**: The random forest is expected to have lower variance than a single decision tree because it averages out the predictions of many trees. However, if the random forest's trees are too simple (e.g., `max_depth` is too low), it might not capture the complexity of the data, leading to underfitting.
- **Balance Between Classes**: Both models use methods to balance the classes. The decision tree directly uses the `class_weight` parameter, while the random forest uses `balanced_subsample`, which looks at the class distribution within each bootstrap sample. The effectiveness of these strategies depends on the specific data distribution.

Considering these points, the decision tree appears to perform better on both the training and validation sets. However, it is essential to also evaluate the models on a test set to get an unbiased estimate of their generalization performance.

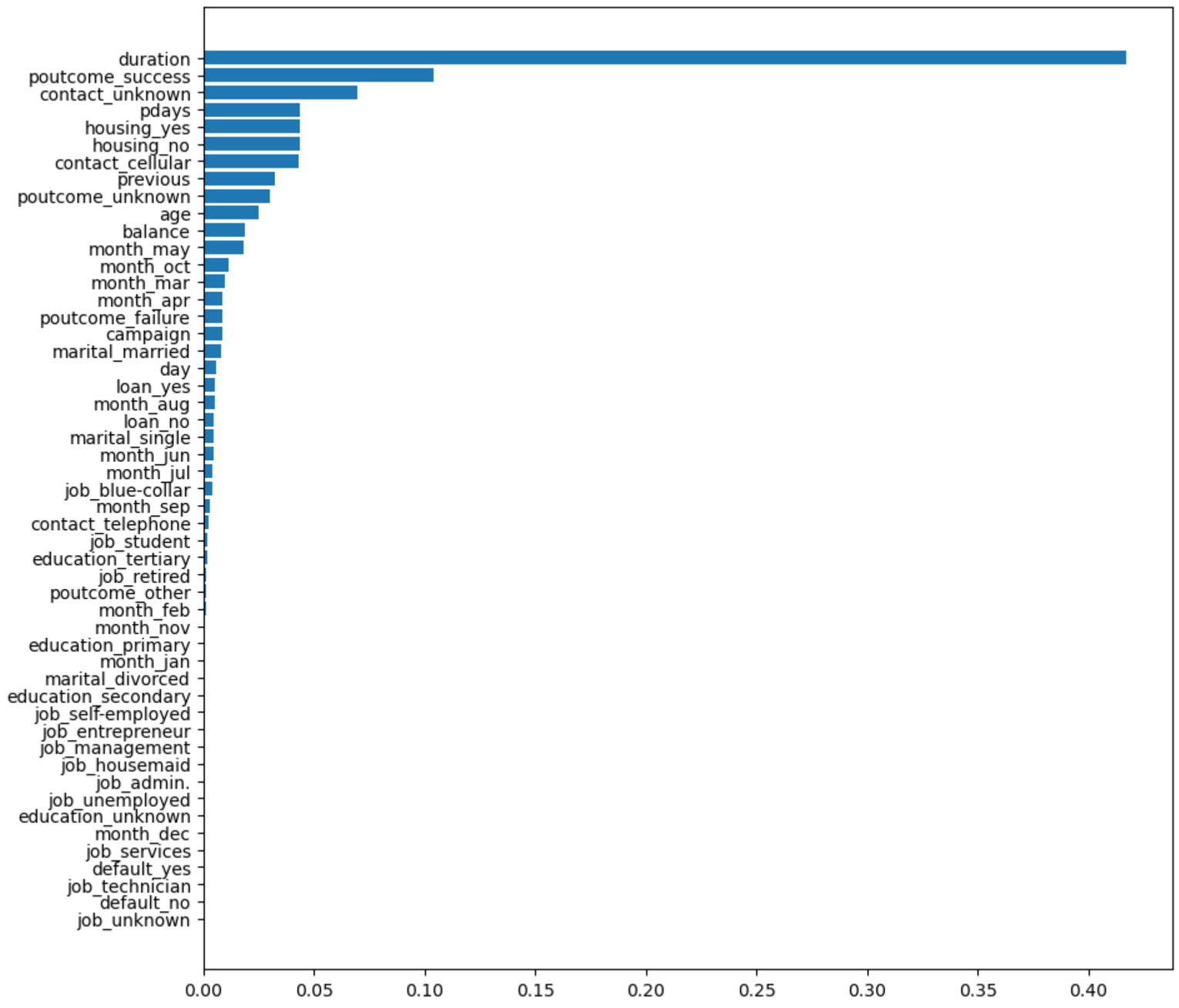
The best course of action would be to:

- Assess the models' performance on a test set.
- Consider further tuning the random forest's hyperparameters, potentially increasing `max_depth` or altering other parameters to capture the data's complexity better.
- Evaluate other performance metrics that might be critical for the problem at hand, such as precision, recall, or the ROC-AUC score.

It's also important to remember that higher complexity models, while potentially offering better performance, can also lead to overfitting and may be harder to interpret. The decision to prefer one model over another should also consider the model's intended use, the importance of interpretability, and the computational resources available for training and prediction.

```
tree_feature_importances = clf.feature_importances_
sorted_idx = tree_feature_importances.argsort()

plt.figure(figsize=(10,10))
plt.barh(dataXExpand.columns[sorted_idx], tree_feature_importances[sorted_idx])
plt.show()
```



- Based on the above figure, do you see any reason to be concerned about the model?
- If the model uses `duration` to predict the target, what can be an issue?

The `duration` feature stands out as being the most important one by a significant margin.

Concerns About the Model Based on Feature Importance:

1. **Over-Reliance on `duration`**: The model might be heavily relying on the `duration` feature to make predictions. If `duration` does not have a causal relationship with the target variable or if it is not available at the time of prediction in a real-world setting, this reliance could be problematic.
2. **Data Leakage**: If `duration` represents the last contact duration and the goal is to predict whether someone will subscribe to a term deposit before calling them, using `duration` might introduce data leakage. Data leakage occurs when information from outside the training dataset is used to create the model. This can cause the model to have an overly optimistic performance during training and validation, but it may perform poorly on real-world data.
3. **Generalization**: A model that depends too much on one feature may not generalize well to new data, especially if the distribution of that feature changes or if that feature is not representative of the underlying process that generates the data.
4. **Feature Engineering**: The other features appear to have significantly lower importance, which might indicate that there's room for improvement in feature engineering. Perhaps combining some features or creating new ones could lead to a more balanced and robust model.

Issues with Using `duration` to Predict the Target:

- **Future Data Availability**: `duration` is likely only known after a call is completed. If the model is intended to predict the outcome before making the call, then `duration` would not be available and thus should not be used as a feature.
- **Causality**: The `duration` might be more of an effect rather than a cause. A successful subscription could naturally lead to longer calls as the details are discussed, rather than longer calls causing the subscription.
- **Practical Use**: For deploying the model in a real scenario, such as deciding whom to call, `duration` would not be usable as a predictive feature.

Best Course of Action:

- **Reassess Feature Relevance**: If `duration` is not appropriate to use as a feature, remove it and retrain the model. This will give you a better understanding of how the other features contribute to the predictions.
- **Domain Knowledge Consultation**: Consult with domain experts to understand whether `duration` is a feature that can or should be used based on the business problem.
- **Model Retraining**: If removing `duration`, the model will need to be retrained and re-evaluated to ensure it still performs adequately.
- **Feature Engineering**: Consider whether additional features could be engineered or whether existing features could be transformed to better capture the underlying relationships in the data.

By addressing these concerns, you can work towards building a model that makes predictions based on more general and causally relevant features, improving the likelihood that the model will perform well both in validation and in practice.

Exercise: Regression Decision Tree

A regression decision tree can also be trained. These are decision trees where the leaf node is a regression function. You will investigate learning regression trees using the boston housing data set from previous labs.

The below code snippet will help get you started. Note that it does not make sense to use entropy for generating splits, so the default method from sklearn will be used. Also note that the DecisionTreeRegressor class uses similar pre-pruning parameters.

```
1  import pandas as pd
```

```
import matplotlib.pyplot as plt
import numpy as np
import sklearn

from sklearn import tree
from sklearn import preprocessing
from sklearn import metrics
from sklearn import model_selection

# Load Boston housing dataset
bostonData = pd.read_csv('./Lab/housing.data.csv', delimiter="\s+")

bostonDataTarget = bostonData['MEDV']
bostonDataAttrs = bostonData.drop(columns='MEDV')
trainY, testY, trainX, testX = model_selection.train_test_split(np.array(bostonDataTarget),np.ar
clfBoston = sklearn.tree.DecisionTreeRegressor(max_depth=5, min_samples_split=5)
clfBoston = clfBoston.fit(trainX, trainY)
predictions = clfBoston.predict(testX)
metrics.mean_squared_error(testY, predictions)

```

```
34.82527594488324
```

- How does the error on the regression decision tree compare to the best results you have found in previous labs?
- Find a good set of pre-pruning parameters that minimises the mean square error

```
from sklearn.model_selection import GridSearchCV

# Define a range of potential values for the pre-pruning parameters
param_grid = {
    'max_depth': [3, 5, 10, None],  # None means the tree can grow as deep as necessary
    'min_samples_split': [2, 10, 20],
    'min_samples_leaf': [1, 5, 10],
    'max_features': [None, 'sqrt', 'log2'],  # None means use all features
}

# Create a DecisionTreeRegressor instance
dt_regressor = sklearn.tree.DecisionTreeRegressor(random_state=0)

# Create the GridSearchCV instance
grid_search = GridSearchCV(dt_regressor, param_grid, cv=5, scoring='neg_mean_squared_error')  # Using negative MSE

# Fit GridSearchCV
grid_search.fit(trainX, trainY)

# Get the best parameters and corresponding MSE
best_params = grid_search.best_params_
best_mse = -grid_search.best_score_  # Negating back to positive MSE

print(f"Best MSE: {best_mse:.2f}")
print(f"Best parameters: {best_params}")

```

```
Best MSE: 19.98
Best parameters: {'max_depth': 10, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 10}
```

The code is using the `GridSearchCV` class from the `model_selection` module of the `sklearn` library to perform a grid search for the best hyperparameters of a decision tree regressor.

Here's a step-by-step explanation of what the code does:

1. `param_grid = {...}`: This dictionary defines the grid of hyperparameters that will be searched. Each key is a hyperparameter name and each value is a list of values that will be tried for that hyperparameter. In this case, the grid includes four hyperparameters: `max_depth`, `min_samples_split`, `min_samples_leaf`, and `max_features`.
2. `dt_regressor = sklearn.tree.DecisionTreeRegressor(random_state=0)`: This line is creating an instance of the `DecisionTreeRegressor` class with a fixed random state to ensure that the results are reproducible.
3. `grid_search = GridSearchCV(dt_regressor, param_grid, cv=5, scoring='neg_mean_squared_error')`: This line is creating an instance of the `GridSearchCV` class. The first argument is the estimator that will be used, the second argument is the grid of hyperparameters that will be searched, the third argument is the number of folds to use for cross-validation, and the fourth argument is the scoring metric that will be used to evaluate the performance of the estimator. In this case, the negative mean squared error is used, which means that the grid search will try to find the hyperparameters that minimize the mean squared error.
4. `grid_search.fit(trainX, trainY)`: The `fit` method is used to perform the grid search. It trains the estimator with each combination of hyperparameters in the grid and evaluates its performance using cross-validation. The combination of hyperparameters that gives the best performance is selected as the best parameters.
5. `best_params = grid_search.best_params_`: This line is retrieving the best parameters found by the grid search.
6. `best_mse = -grid_search.best_score_`: This line is retrieving the best score found by the grid search and negating it to get the mean squared error (since the grid search was using the negative mean squared error as the scoring metric).