Module 6 exercises: Classification

The problem

- Loan data for every customer who borrowed £1000 for 12 months.
- Examples/cases = row = single customer/loan.
- Features = columns = fields = characteristics of customer/loan.
- Target: the characteristic/column you're trying to predict/understand.
    - For this problem, it is **default**, which takes a value of 0 if the applicant paid back their loan, and 1 if they did not.
- Problem: what model best describes the relationship of the features to the target?

Step 1: Learn pattern (model) from data which describes features relationship to target.

Step 2: Use pattern to guess unknown target from known features.

Exercise 1

1. Import the following packages:
    a. pandas
    b. matplotlib
    c. numpy
    d. seaborn

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

2. Load the following data file into a `DataFrame`:
    a. `loan_data.csv`

```
# Read in the loans data and assign it to a dataframe
df = pd.read_csv('data/loan_data.csv')

# View a smaple of the data
df.sample(3)
```

	ID	Income	Term	Balance	Debt	Score	Default
42	841	20600.0	Short Term	1640.0	91.0	517.0	False
823	876	44000.0	Long Term	1550.0	0.0	950.0	False
846	181	40700.0	Long Term	1000.0	0.0	664.0	False

Exercise 2 – EDA

1. Familiarise yourself with the dataset. Seek to understand:
    - the types of data.
    - distributions of features.
    - obvious patterns.
    - errors, nulls, outliers.

```
# viewing the info
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 856 entries, 0 to 855
Data columns (total 7 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   ID       856 non-null    int64  
 1   Income   856 non-null    float64
 2   Term     856 non-null    object 
 3   Balance  856 non-null    float64
 4   Debt     856 non-null    float64
 5   Score    836 non-null    float64
 6   Default  856 non-null    bool   
dtypes: bool(1), float64(4), int64(1), object(1)
memory usage: 41.1+ KB
```

```
# Dropping unecessary columns, ID is of no use to us
df.drop(columns='ID', inplace = True)
```

```
# Summary Statistics of numerical columns
df.describe().round()
```

	Income	Balance	Debt	Score
count	856.0	856.0	856.0	836.0
mean	29882.0	1214.0	644.0	451.0
std	13976.0	588.0	1150.0	269.0
min	11800.0	140.0	0.0	0.0
25%	19800.0	910.0	0.0	243.0
50%	22900.0	1120.0	65.0	376.0
75%	39025.0	1370.0	959.0	647.0
max	86000.0	6020.0	12891.0	1000.0

```
# Summary of categorical colums
df['Term'].value_counts()
```

```
Term
Short Term    584
Long Term     272
Name: count, dtype: int64
```

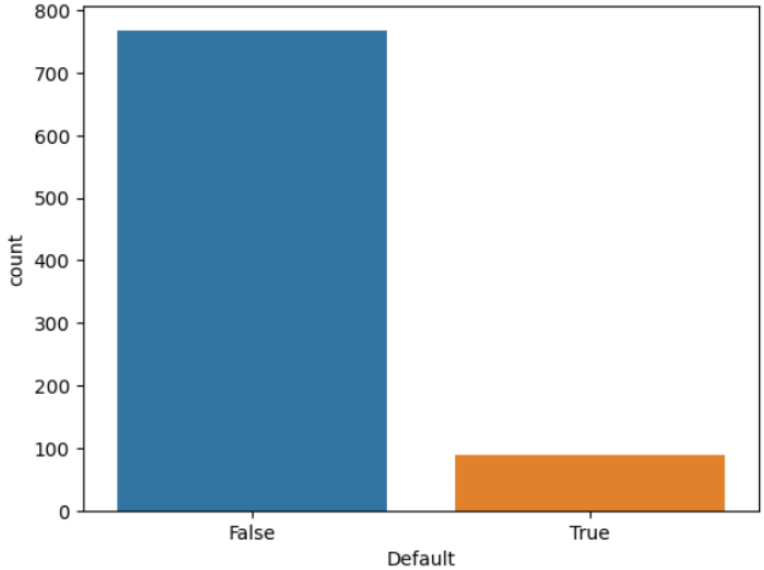
Mostly short term loans

Examining our Target variable

```
# Counts of the target variable
df.Default.value_counts()
```

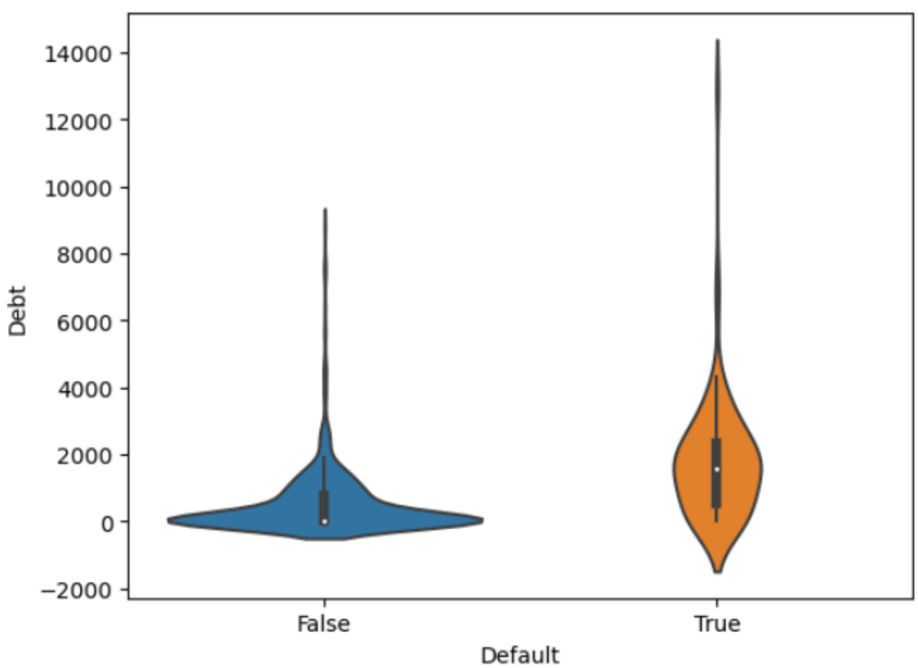
```
Default
False    768
True      88
Name: count, dtype: int64
```

```
# Visualisation of this
sns.countplot(x='Default', data = df);
```



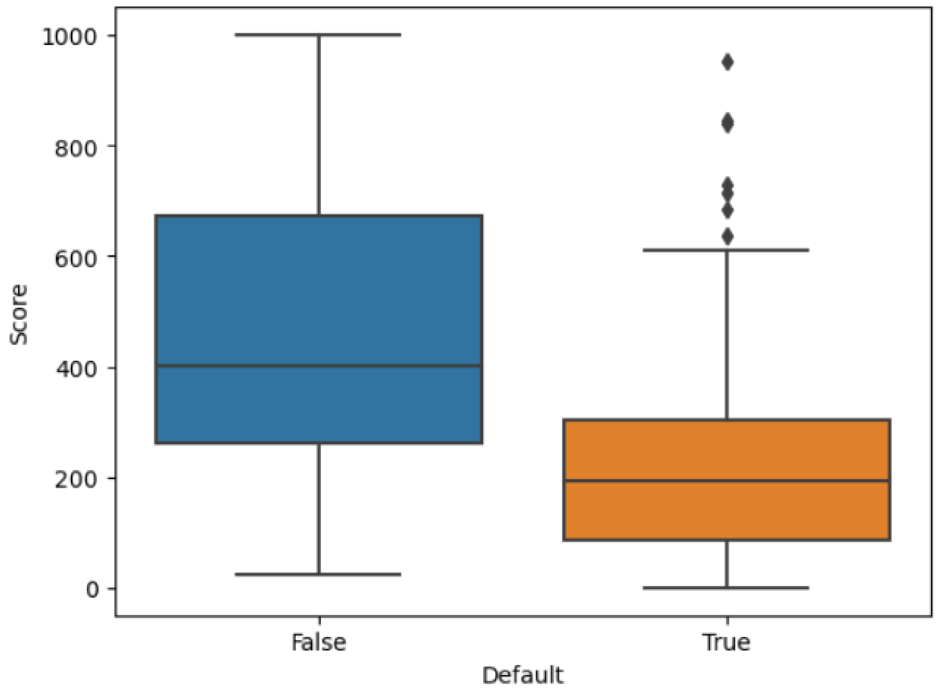
We have heavily imbalanced data, this is an issue.
We will continue with a vanilla model to see its perfromance and will investigate methods to help with this

```
# Violin Plot for debt
sns.violinplot(x="Default",
               y="Debt",
               data=df);
```



```python
# Box plot of credit score\
sns.boxplot(x="Default", y="Score", data=df)
```

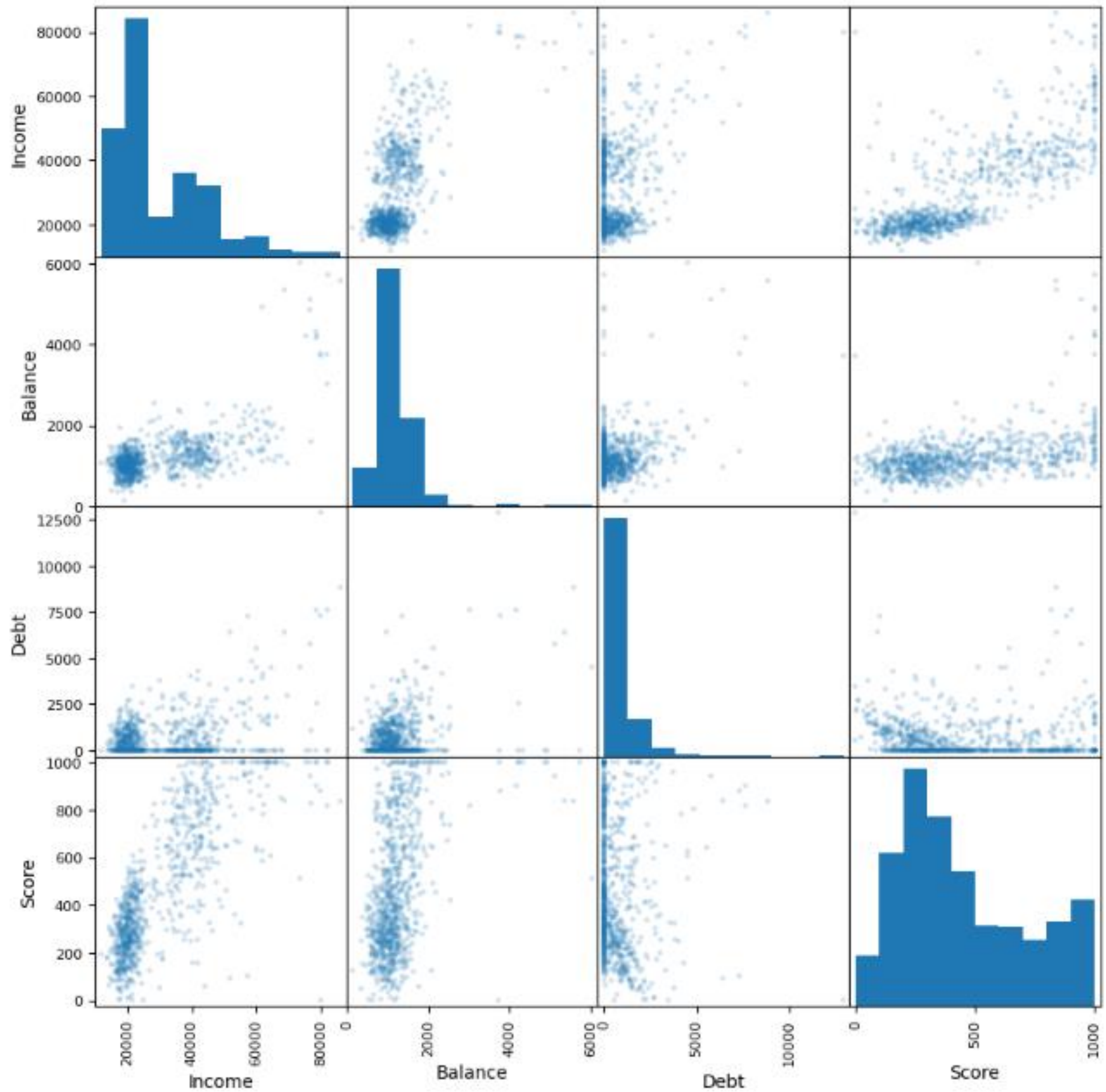
```
<Axes: xlabel='Default', ylabel='Score'>
```




```python
# Calling the correlation matrix on all numeric columns
sns.heatmap(df.iloc[:,:-1].corr(numeric_only=True), cmap='coolwarm', center=0.0, annot=True)
```

```
<Axes: >
```

```
pd.plotting.scatter_matrix(df.iloc[:,:-1], alpha=0.2,figsize =(10,10));
```



Exercise 3 - Data preparation

Prepare the dataset for modelling by performing the following steps:

1. Remove or impute null values (with justification).

```
# Identify our null values
df.isna().sum()
```

```
Income       0
Term         0
Balance      0
Debt         0
Score       20
Default      0
dtype: int64
```

```
# Dropping the 20 rows with Nan values
df.dropna(inplace = True)
df.describe().round()
```

	Income	Balance	Debt	Score
count	836.0	836.0	836.0	836.0
mean	29907.0	1219.0	648.0	451.0
std	14021.0	593.0	1154.0	269.0
min	11800.0	140.0	0.0	0.0
25%	19775.0	910.0	0.0	243.0
50%	22900.0	1120.0	71.0	376.0
75%	39025.0	1380.0	966.0	647.0
max	86000.0	6020.0	12891.0	1000.0

2. Encode the target variable using `LabelEncoder`.

```python
from sklearn import preprocessing
# Calling the label encoder
le = preprocessing.LabelEncoder()

# Fitting it to the target columns
le.fit(df['Default'])

# Creating a new column which will hold the old labels for us to inspect
# This column in unecessary and will be dropped
# It is just for us to examine
df['old_labels'] = df['Default']

# Transforming the target column
df['Default'] = le.transform(df.Default)
```

```python
# We can examine the new column created alongside the old labels
df.sample(10)
```

	Income	Term	Balance	Debt	Score	Default	old_labels
560	44900.0	Short Term	1190.0	419.0	761.0	0	False
457	18600.0	Short Term	890.0	1096.0	132.0	0	False
518	33800.0	Long Term	1140.0	22.0	437.0	0	False
143	19200.0	Short Term	640.0	322.0	130.0	0	False
194	18500.0	Short Term	890.0	0.0	143.0	0	False
490	17000.0	Short Term	360.0	1537.0	49.0	1	True
403	24600.0	Short Term	1300.0	0.0	303.0	0	False
577	37900.0	Short Term	1570.0	0.0	988.0	0	False
704	29100.0	Short Term	1730.0	325.0	640.0	0	False
576	32700.0	Short Term	1120.0	0.0	619.0	0	False

```python
# We can drop the old labels column now
df.drop('old_labels', axis=1, inplace=True);
```

3. Encode categorical features using either `OneHotEncoder` or `OrdinalEncoder` (with justification).

```python
# Importing the required method
from sklearn.preprocessing import OrdinalEncoder

# Calling the encoder and specifying the order to encode
enc = OrdinalEncoder(categories = [['Short Term', 'Long Term']])

# Fitting it to a created column for the data,'Term_ordinal'
df['Term_ordinal'] = enc.fit_transform(df['Term'].values.reshape(-1, 1))
```

```python
df.head()
```

	Income	Term	Balance	Debt	Score	Default	Term_ordinal
0	17500.0	Short Term	1460.0	272.0	225.0	0	0.0
1	18500.0	Long Term	890.0	970.0	187.0	0	1.0
2	20700.0	Short Term	880.0	884.0	85.0	0	0.0
4	24300.0	Short Term	1260.0	0.0	495.0	0	0.0
5	22900.0	Long Term	1540.0	1229.0	383.0	0	1.0

4. Split the data into training and testing sets. Recall, **default** is our target.

```python
from sklearn.model_selection import train_test_split

# we make our test set
X_train, X_test, y_train, y_test = train_test_split(df[['Income',
                                                        'Balance',
                                                        'Debt',
                                                        'Score',
                                                        'Term_Long Term',
                                                        'Term_Short Term',
                                                        'Term_ordinal']],
                                                    df['Default'],
                                                    test_size=0.3,
                                                    random_state=1)
```

5. Decide whether to scale (normalise/standardise) numeric features.

```python
# Assigning the columns to scale to a variable
cols_to_scale = ['Income','Balance','Debt','Score']
```

```python
# importing the package
from sklearn.preprocessing import StandardScaler

# Calling the method twice, once for the features and once for the target
scaler =  StandardScaler()

# Fitting the scaler on the training features and applying it to both the training and test data
# Note how the Scaler is only fit to the training set, but then applied to both
X_train[cols_to_scale] = scaler.fit_transform(X_train[cols_to_scale])
X_test[cols_to_scale] = scaler.transform(X_test[cols_to_scale])
```

Exercise 4 – Building your first model

1. After preparing the dataset, we can it our first classification model. Import the `LogisticRegression` model from scikit-learn's `linear_model` module.

```python
from sklearn.linear_model import LogisticRegression
```

2. Create a `LogisticRegression` model, then use the `fit` method to train the model on `X_train` and `y_train`.

```python
# Defining which columns we will use as inputs to our model
cols = ['Income',
        'Balance',
        'Debt',
        'Score',
        'Term_ordinal']
```

```python
lr = LogisticRegression()

lr.fit(X_train[cols], y_train)
```

```
▾ LogisticRegression
LogisticRegression()
```

3. Generate predictions from the model for `X_test` using the `predict` method, storing them in the variable `y_pred_lr`.

```python
y_pred_lr = lr.predict(X_train[cols])
```

Exercise 5 – Building a Decision Tree

1. Using the same approach as above, build a `DecisionTreeClassifier` model with the following parameter settings:

- max_depth=2
- min_samples_leaf=20
- random_state = 42

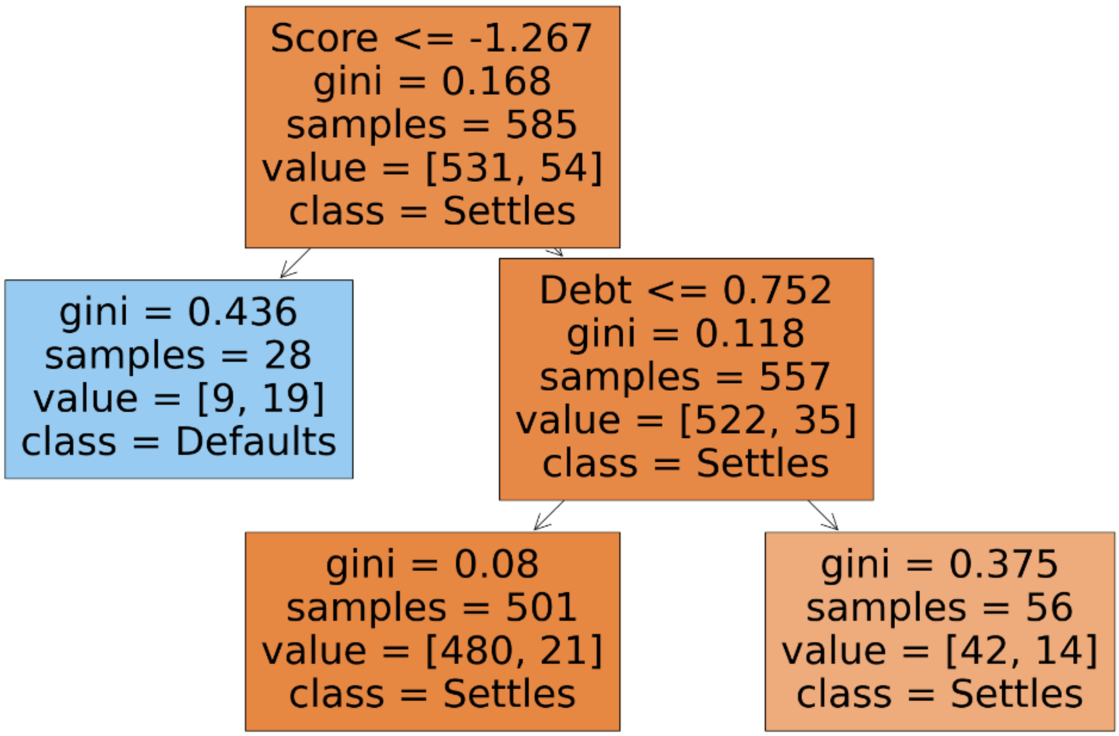
The model can be obtained from `sklearn.tree`.

```
# Importing Decision Tree classifier
from sklearn.tree import DecisionTreeClassifier, plot_tree


# calling the decision tree method
# setting max_depth to 2
# and min sample per leaf at 20
# this mean ther must be at least 20 samples in each leaf
dt = DecisionTreeClassifier(max_depth=2,
                            min_samples_leaf=20,
                            random_state = 42)


# fitting the decision tree onto the training data
dt.fit(X_train[cols], y_train)


y_pred_dt = dt.predict(X_test[cols])
```

2. Use the `plot_tree` function from `sklearn.tree` to visualise the decision tree you have built. You can copy the sample code below if required:

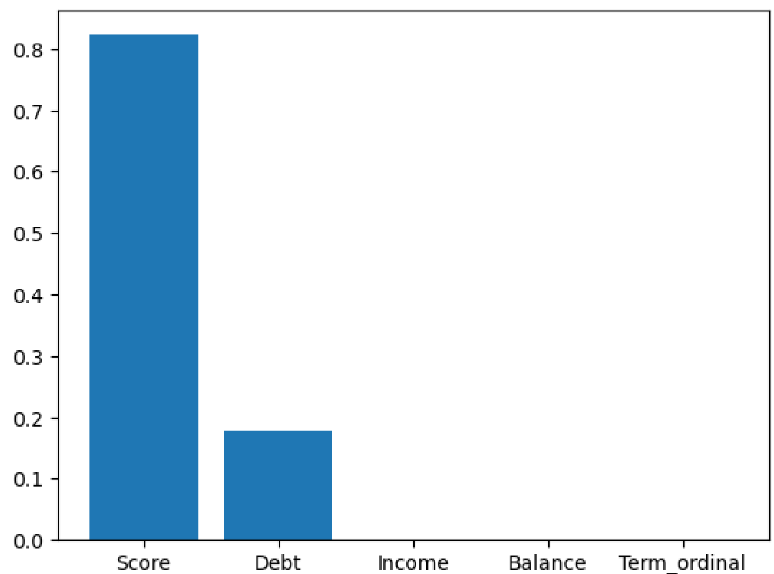
```
fig = plt.figure(figsize=(25,15))

_ = tree.plot_tree(dt,
              feature_names = X_train[cols].columns,
               class_names = ['Settles','Defaults'],
               filled=True
              )
```

```python
# plotting the decision tree
# This plot shows us how the tree was built
# the data at each node
# the Gini values

# from sklearn import tree
# from matplotlib import pyplot as plt

fig = plt.figure(figsize=(25,15))

_ = plot_tree(dt,
              feature_names = list(X_train[cols].columns),
               class_names = ['Settles','Defaults'],
               filled=True
              )
```



3. If you have time, examine the `feature_importances_` attribute of the model. The name of each feature listend can be obtained from `feature_names_in_`.

```python
# extracting the feature importance of the model
# can be acessed by df.feature_importances_

# zipping together the feature anmes and thier scores
f_i = list(zip(dt.feature_names_in_, dt.feature_importances_))

# sorting by scores, highest first
f_i.sort(key = lambda x : x[1],reverse=True)

# plotting these scores
plt.bar([x[0] for x in f_i],[x[1] for x in f_i])

plt.show()
```



Exercise 6 – Building a Random Forest

1. Using the same approach as above, build a `RandomForestClassifier` model with the following parameter settings:

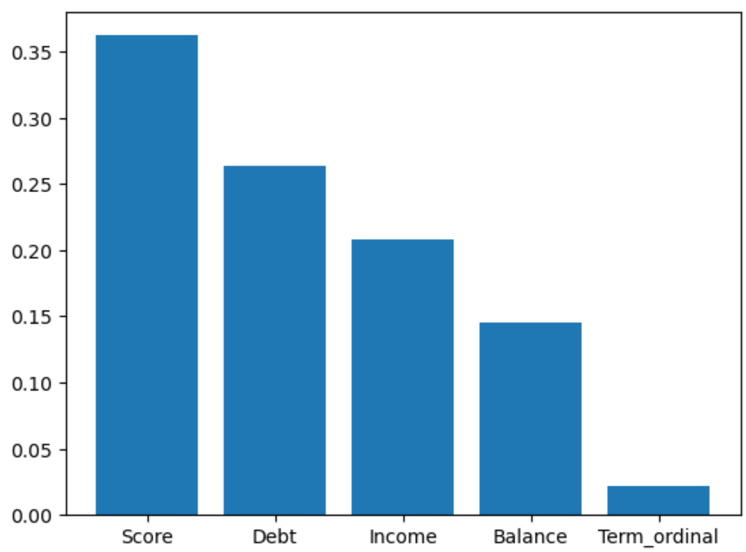
    - random_state = 42

    The model can be obtained from `sklearn.ensemble`.

```
# Importing packages
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(random_state = 42)

rf.fit(X_train[cols], y_train)

y_pred_rf = rf.predict(X_test[cols])
```

2. If you have time, examine the `feature_importances_` attribute of the model. The name of each feature listend can be obtained from `feature_names_in_`.

```
f_i = list(zip(rf.feature_names_in_,rf.feature_importances_))
f_i.sort(key = lambda x : x[1],reverse=True)
plt.bar([x[0] for x in f_i],[x[1] for x in f_i])

plt.show()
```



Exercise 7 – Evaluating your models

1. Display sklearn's `classification_report` for your models, obtained from the `metrics` module.
2. Interpret the following elements of it:
   - accuracy
   - precision
   - recall

- f1-score

```python
from sklearn.metrics import classification_report

print("Logisitic Regression")
print(classification_report(y_test,
                            y_pred_lr,
                            target_names=["Settles", "Defaults"]))
print("Decision Tree")
print(classification_report(y_test,
                            y_pred_dt,
                            target_names=["Settles", "Defaults"]))
print("Random Forest")
print(classification_report(y_test,
                            y_pred_rf,
                            target_names=["Settles", "Defaults"]))
```

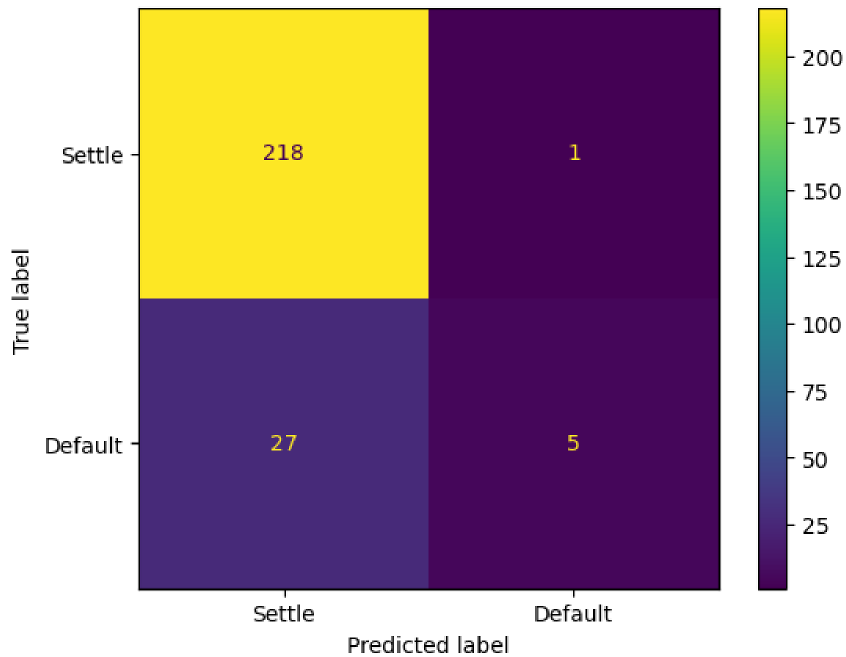
```
Logisitic Regression
              precision    recall  f1-score   support

     Settles       0.89      1.00      0.94       219
    Defaults       0.83      0.16      0.26        32

    accuracy                           0.89       251
   macro avg       0.86      0.58      0.60       251
weighted avg       0.88      0.89      0.85       251

Decision Tree
              precision    recall  f1-score   support

     Settles       0.90      0.99      0.94       219
    Defaults       0.80      0.25      0.38        32

    accuracy                           0.90       251
   macro avg       0.85      0.62      0.66       251
weighted avg       0.89      0.90      0.87       251

Random Forest
              precision    recall  f1-score   support

     Settles       0.92      1.00      0.96       219
    Defaults       1.00      0.41      0.58        32

    accuracy                           0.92       251
   macro avg       0.96      0.70      0.77       251
weighted avg       0.93      0.92      0.91       251
```

3. Construct a confusion matrix using the following code:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

predictions = lr.predict(X_test[cols])

cm = confusion_matrix(y_test,
                      predictions)

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=('Settle','Default'))

disp.plot();
```

How well does each model perform? Would you use it?

```
# Displaying precision and recall figures
print(classification_report(y_test, predictions, target_names=["Sett
les", "Defaults"]))

# PLotting the confusion matrix
predictions = rf.predict(X_test[cols])

cm = confusion_matrix(y_test,
                      predictions,
                      labels=rf.classes_)

disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=('Settle','Default')
                             )

disp.plot();
```



Exercise 8 – Feature Engineering and Hyperparameter Tuning

1. Experiment with features and hyperparameters to see whether you can improve the models you have built.

```
clf = lm.LogisticRegression(penalty=None,
                            solver='lbfgs',
                            max_iter=1000,
                            class_weight="balanced"
                           )

clf.fit(X_train[cols], y_train)
```

```
▾                    LogisticRegression
LogisticRegression(class_weight='balanced', max_iter=1000, penalty=None)
```

```
print(classification_report(y_test,
                            clf.predict(X_test[cols]),
                            target_names=["Settles", "Defaults"]))
```

```
              precision    recall  f1-score   support

     Settles       0.95      0.84      0.89       219
    Defaults       0.40      0.72      0.51        32

    accuracy                           0.82       251
   macro avg       0.67      0.78      0.70       251
weighted avg       0.88      0.82      0.84       251
```

Extension (to be completed after Module 7: Model Selection and Evaluation)

Exercise 9 – ROC Curve

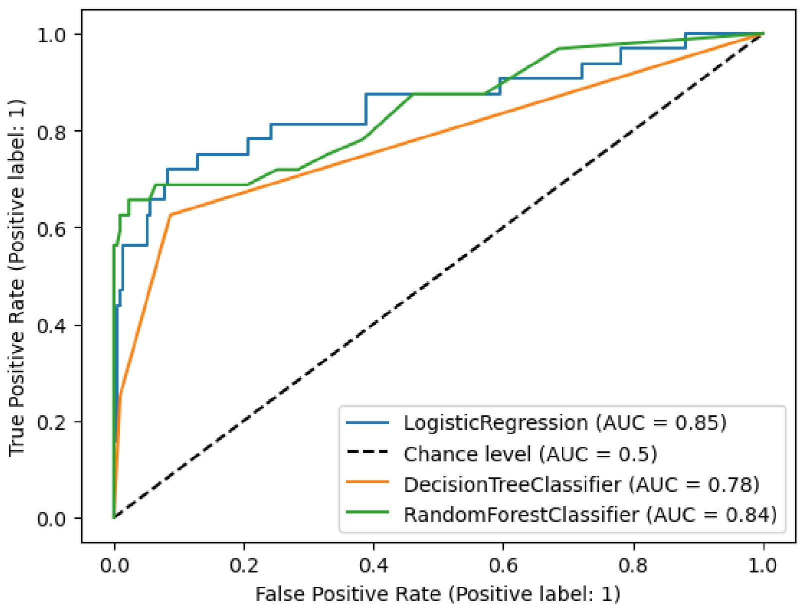
1. Using the below code, compare each model using an ROC curve.

Note: The code will only work if you create a list called models which contains each model you have built.

```
from sklearn.metrics import RocCurveDisplay
for i, model in enumerate(models):
    if i == 0:
        plot = RocCurveDisplay.from_estimator(model,
                                              X_test,
                                              y_test,
                                              plot_chance_level=True)
        axes = plot.ax_
    else:
        RocCurveDisplay.from_estimator(model,
                                       X_test,
                                       y_test,
                                       plot_chance_level=False,
                                       ax=axes)
```

Which model performs best?

```python
from sklearn.metrics import RocCurveDisplay

models = [lr, dt, rf]

for i, model in enumerate(models):
    if i == 0:
        plot = RocCurveDisplay.from_estimator(model,
                                              X_test[cols],
                                              y_test,
                                              plot_chance_level=True)
        axes = plot.ax_
    else:
        RocCurveDisplay.from_estimator(model,
                                       X_test[cols],
                                       y_test,
                                       plot_chance_level=False,
                                       ax=axes)
```



Logistic regression and Random Forest both appear to be most performant.