A. Explanation of entropy in classification

In a binary classification problem, the entropy S of a partition Π is defined as

$$S = -p\log(p) - (1-p)\log(1-p),$$

where p is the fraction of elements in Π that are of class 1, and $1-p$ is the fraction of class 0. We differentiate the entropy with respect to p to find the stationary points:

$$S'(p) = -\log(p) + \log(1-p) = 0.$$

Equating the logarithms, we must have $p = 1-p$, so $p = 1/2$ is a stationary point. Differentiating once again gives

$$S''(p) = -\frac{1}{p} - \frac{1}{1-p} < 0,$$

since $0 \leq p \leq 1$. Therefore, $p = 1/2$ yields a maximum for the entropy function. As a consequence,

(a) High entropy means the partitions are pure. This statements is **false**. Entropy reaches a maximum when the number of elements of different classes is balanced within the partition. This is equivalent to the partition being impure. When a partition is pure, we expect $p = 0$ or $p = 1$. In either case,

$$\lim_{p\to 0} S(p) = 0 = \lim_{p\to 1} S(p).$$

(b) High entropy means the partition are impure. This statement is **true.** As shown above, when partitions are impure (classes are closed to balanced), the entropy achieves a maximum.

B. Feature selection using the funnelling approach

In this report, we aim to predict up and down movements of an asset's price from past information. The ticker for the chosen stock is `CSU.TO`, associated with Constellation Software, a Canadian company that has seen unprecedented growth in the last decade. The 5-year data (up until May 9th 2025) was retrieved using the `yfinance` library and stored in the attached `csu.csv` file. It is then imported as a pandas dataframe named `csu`.

```
csu = pd.read_csv('csu.csv')
```

The first step in our machine learning model for stock price direction prediction is to select features that could potentially be correlated with the price movements. While many feautures are proposed, the most significant ones will be chosen using a funnelling approach involving filtering, wrapping, and embedding methods. The candidate features are listed below.

- `Close`: The price of one share at the time the market closes.
- `High`: The maximum share price in a given day.
- `Low`: The minimum share price in a given day
- `Open`: The price of one share at the time the market opens.
- `Volume`: The number of share traded in a given day.
- `Log Returns`: The logarithms of the quotient between the closing prices on the present day and the previous day.
- `High-Low`: The difference between `High` and `Low`.
- `Open-Close`: The difference between `Open` and `Close`.
- `d-day Momentum`: The difference in the closing prices on the present day and d days ago.
- `d-day Moving Average`: The average closing share price calculated using information from the last d days.
- `d-day Volatility`: The volatility of the closing share prices calculated using information from the last d days.

- `d-day Quarter Upper Band`: One quarter of the upper Bollinger band calculated using information from the last d days.

We will consider values for $d = 5, 10, 20$ to calculate momentum, volatility, moving averages and Bollinger bands. All of these are calculated and added into the `csu` data frame.

```
csu["Log Returns"] = np.log(csu["Close"]/csu["Close"].shift(1))
csu["High-Low"] = csu["High"]- csu["Low"]
csu["Open-Close"] = csu["Open"] - csu["Close"]
csu["5d Momentum"] = csu["Close"] - csu["Close"].shift(5)
csu["5d Moving Average"] = csu["Close"].rolling(5).mean()
csu["5d Rolling Volatility"] = csu["Close"].rolling(5).std()
csu["5d Quarter Upper Band"] = csu["5d Moving Average"] + 0.5*csu["5d Rolling Volatility"]
csu["10d Momentum"] = csu["Close"] - csu["Close"].shift(10)
csu["10d Moving Average"] = csu["Close"].rolling(10).mean()
csu["10d Rolling Volatility"] = csu["Close"].rolling(10).std()
csu["10d Quarter Upper Band"] = csu["10d Moving Average"] + 0.5*csu["10d Rolling Volatility"]
csu["20d Momentum"] = csu["Close"] - csu["Close"].shift(20)
csu["20d Moving Average"] = csu["Close"].rolling(20).mean()
csu["20d Rolling Volatility"] = csu["Close"].rolling(20).std()
csu["20d Quarter Upper Band"] = csu["20d Moving Average"] + 0.5*csu["20d Rolling Volatility"]
```

To implement feature selection, we start by assigning labels to the stock price movements. The label `1` is assigned if the closing price the day after is greater than the closing price on the current date. To discard small increases, we calculate the logarithmic returns and find the 25-th percentile of the positive values,

```
quantile_25 = csu[csu["Log Returns"] > 0]["Log Returns"].quantile(0.25)
```

which was calculated to be

$$Q_{25} = 0.47\%.$$

Any logarithmic returns smaller than Q_{25} are not classified as an upwards movement and assigned the 0 label. All downwards movements are also labelled 0, as shown in the following script

```
csu = csu.dropna() # Drop all NaN values
y = np.where(csu["Log Returns"].shift(-1) > quantile_25, 1, 0)
```

B.1 Filtering methods

The funnelling approach for feature selection begins with the implementation of filtering methods. A first filter must examine multicollinearity. If one feature can be expressed as a linear combination of others, they are redundant and some can be discarded. Is is evident that features such as `Open`, `Close`, and `Open - Close` are are perfectly correlated. Preliminarily, we can discard the `Close` and `Low` features. A similar argument can be made for the Bollinger bands, which are calculated from the volatility. Thus, we also abandon the `Quarter Upper Band` features. We also expect the 5, 10, and 20 day momenta, averages, and volatilities to be somewhat correlated. Indeed, we will use feature selection to decide a priori which time period has a better predictive power. We will not combine features from different time windows.

Given this initial naive filtering, let us now verify the correlations between the remaining features using the Variance Inflation Factor (VIF). We use the code provided in the solutions of the Exercise Sheet in Course 4's Supervised Learning I Module.

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

def vif(X):
    xs = scaler.fit_transform(X)
    vif = pd.DataFrame()
    vif["Features"] = X.columns
    vif["VIF Factor"] = [variance_inflation_factor(xs, i) for
```

```
                    i in range(xs.shape[1])]
    return vif
```

A first analysis showed that the `High` and `Open` features are also highly correlated with each other and also with the expected moving average. We discard these two features. The three final tables with VIF factors for the three different time periods ($d = 5, 10, 20$ days) are shown below.

Features	VIF factor	Features	VIF Factor	Features	VIF Factor
Volume	1.096985	Volume	1.099393	Volume	1.096819
Log Returns	2.623314	Log Returns	2.574904	Log Returns	2.556840
High-Low	2.182529	High-Low	1.853172	High-Low	1.756555
Open-Close	2.612904	Open-Close	2.599055	Open-Close	2.577395
5d Momentum	1.265990	10d Momentum	1.195735	20d Momentum	1.346292
5d Moving Avg.	1.778037	10d Moving Avg.	1.997680	20d Moving Avg.	2.251315
5d Volatility	1.939405	10d Volatility	1.848624	20d Volatility	2.115165

As we see, the multicollinearity in this set of features is not extremely high. Further filtering can be done via Analysis of Variance (ANOVA) methods, which are known to be better suited for binary classification given numerical features. These find correlations of the features and the associated class via an univariate F-test, returning scores representing the significance of each feature. The specific method `SelectKBest` is native to `scikit-learn`. Based on the exercise sheet:

```
X2_df = X_df[["Volume", "Log Returns", "High-Low", "Open-Close",
"5d Momentum", "5d Moving Average", "5d Rolling Volatility",
"10d Momentum", "10d Moving Average", "10d Rolling Volatility",
"20d Momentum", "20d Moving Average", "20d Rolling Volatility"]]
X2 = X2_df.values

skb = SelectKBest(f_regression, k=6)
skb.fit(X2,y)

for f, s in zip(X2_df.columns, skb.scores_):
    print(f'F-score: {s:0.4} for feature {f}')
```

The results of this analysis are shown in the table below.

Features	F-score	Features	F-score
Volume	1.201	10d Momentum	2.296
Log Returns	6.132	10d Moving Avg.	0.4087
High-Low	0.2306	10d Volatility	2.774
Open-Close	6.36	20d Momentum	2.199
5d Momentum	1.725	20d Moving Avg.	0.4795
5d Moving Avg.	0.3748	20d Volatility	1.557
5d Volatility	0.1578		

We observe that the `High-Low` has low predictive power according to the F-score metric. Additionally, 10-day period metrics seem to score better on average. Overall, moving averages do not seem to be very predictive. We will keep these features for the time being and confirm whether they should be discarded using further methods.

B.2 Wrapper Methods

The method we will use for classifying up and down trends is the Gradient Boosting Classifying. Thus, we implement a wrapper method known as recursive feature elimination (RFE) based on this classifier to further narrow down features. RFE uses the underlying classifier using different subsets of features and measures the classfication accuracy. Then, it ranks the first n features that resulted in higher

accuracies. The method is build into `scikit-learn` as `feature_selection.RFE`. Drawing on the aforementioned exercise sheet, we adapt the code for our purposes.

```
rfe = RFE(GradientBoostingClassifier(max_depth = 3), n_features_to_select = 6, step = 1)

rfe.fit(X2, y)

list(zip(X2_df.columns, rfe.ranking_))
```

The output is

```
[('Volume', 1),
 ('Log Returns', 1),
 ('High-Low', 1),
 ('Open-Close', 3),
 ('5d Momentum', 4),
 ('5d Moving Average', 6),
 ('5d Rolling Volatility', 7),
 ('10d Momentum', 1),
 ('10d Moving Average', 8),
 ('10d Rolling Volatility', 2),
 ('20d Momentum', 1),
 ('20d Moving Average', 5),
 ('20d Rolling Volatility', 1)]
```

A lower number represents a higher ranking. Here we made the decision now to focus on 6 features. For $n = 6$, we obtain the features selected by RFE are

- `Volume`
- `Log Returns`
- `High-Low`
- `10 day momentum`
- `20 day momentum`
- `20 day volatility`

`Open - Close` and `10 day volatility` are also rated relatively high. Combining these conclusion with out previous findings, we make the following decisions:

1. We keep 10 day momentum and 10 day volatility as features. 20 day metrics are ranked similarly, but they perform slightly worse with ANOVA methods.
2. We keep `Volume` and `Log Returns`, which have consistenly received high scores.
3. To complete the set of six features, we keep `Open-Close` and `High-Low`, which are next in the ranking.
4. We discard moving averages as a predicitive feature.

B.3 Embedded methods

To further confirm our choice of features, we use an embedded method known as Feature Importance. It calculates a hierarchical scoring method index by running the classifier using different thresholds. This is implemented on `scikit-learn` as the `feature_importances_` attribute of the `GradientBoostingClassifier`.

```
X3_df = X_df[["Volume", "Log Returns", "High-Low", "Open-Close", "10d Momentum",
"10d Rolling Volatility"]]
scaler = StandardScaler()
X3 = X3_df.values
X3 = scaler.fit_transform(X3)

X_train, X_test, y_train, y_test = train_test_split(X3, y,
test_size = 0.2, shuffle = False)
gbr = GradientBoostingClassifier(max_depth = 3)
```

```
gbr.fit(X_train, y_train)
list(zip(X3_df.columns, gbr.feature_importances_))
```

The scores for the six selected features are shown in the table below.

Features	Importance
Volume	0.147
Log Returns	0.156
High-Low	0.169
Open-Close	0.150
10d Momentum	0.241
10d Volatility	0.136

The Feature Importance method classifies 10 day momentum as the most relevant features. All other chosen features score lower, but they are assigned similar importances.

In light on the results of this funnelling approach, we keep the following features to build the classifying model:

- Volume
- Log Returns
- High-Low
- Open-Close
- 10-day Momentum
- 10-day Volatility

C. Model Building, Tuning, and Evaluation

C.1 Build a model to predict positive market moves.

As explained in the previous section, the aim of our Machine Learning moel is to predict uptrends, where an uptrend is defined as a logarithmic return greater than $Q_{25} = 0.47\%$. A certain day is assigned the label 1 if the next day is an uptrend. With the features chosen, we display the first few data points.

	Volume	Log Returns	High-Low	Open-Close	10d Momentum	10d Rolling Volatility	Signal
20	115500	-0.002242	27.841477	-13.732765	-98.038696	19.607984	1
21	58500	0.008250	30.154972	-23.857037	-14.000122	19.710163	0
22	53600	-0.021873	26.615519	9.511172	-57.126831	22.297274	0
23	46400	-0.007011	46.240826	28.642256	-93.035767	18.954984	1
24	44400	0.007295	77.691049	-20.376963	-40.091186	19.533380	1

The model requested is Gradient Boosting, which initially uses decision trees of depth `max_depth`, then uses the gradient descent of the loss function with a given `learning_rate` to correct the error made by the trees. There are `n_estimators` "boosting" steps to further correct the error. First, we run an iteration with the following hyperparameter choices:

- `max_depth` = 3
- `learning_rate` = 0.1 (default for built-in `GradientBoostingClassifier`)
- `n_estimators` = 100 (default)

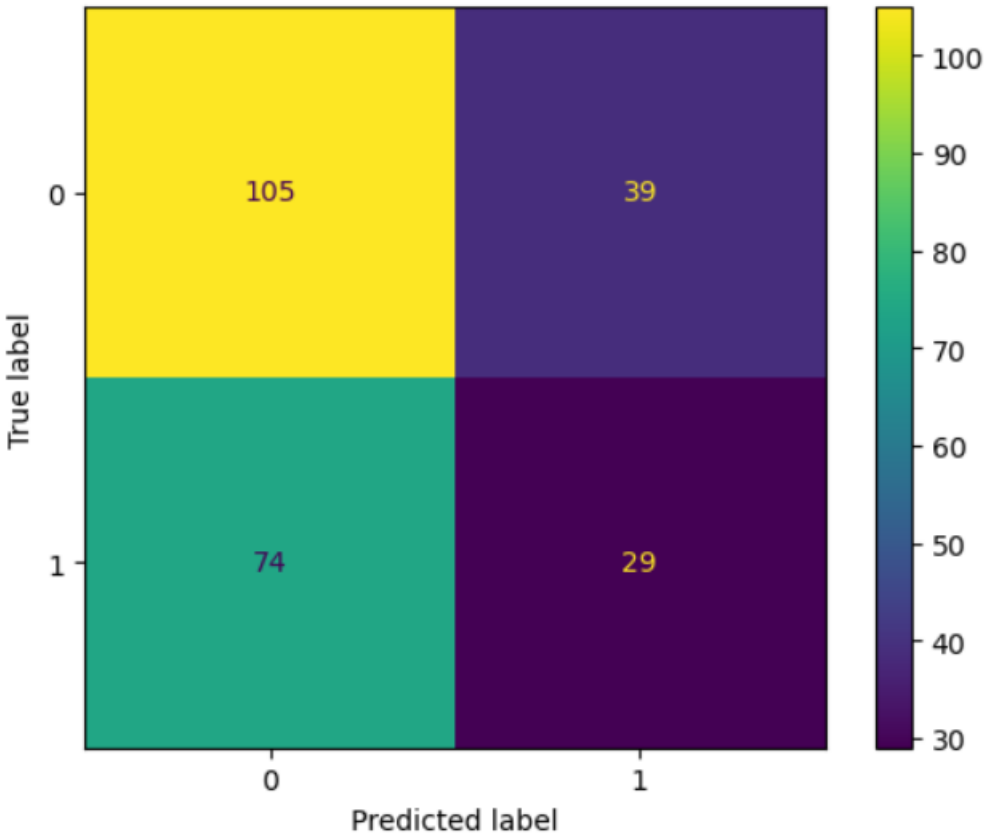
This model was, in fact, already implemented in section B.3. Let us use it to predict the labels of the test set `y_test`. Then we evaluate the recall, defined as the ratio of true positives over total samples labelled classified as positive.

```
y_pred = gbr.predict(X_test)

acc_train = recall_score(y_train, gbr.predict(X_train))
acc_test = recall_score(y_test, y_pred)

print(f'Train Recall: {acc_train:0.4}, Test Recall: {acc_test:0.4}')
```

```
Train Recall: 0.5455, Test Recall: 0.2816
```

The reason we prefer the recall as a performance metric is because we will consider a trading strategy in which an investor buys a share if the model predicts the price will go up, and holds otherwise. For such a scheme, we would like to minimize the number of incorrect buy signals. With this arbitrarily chosen hyperparameters, the recall on the test set is 0.28. We can further evaluate the model using the confusion matrix.

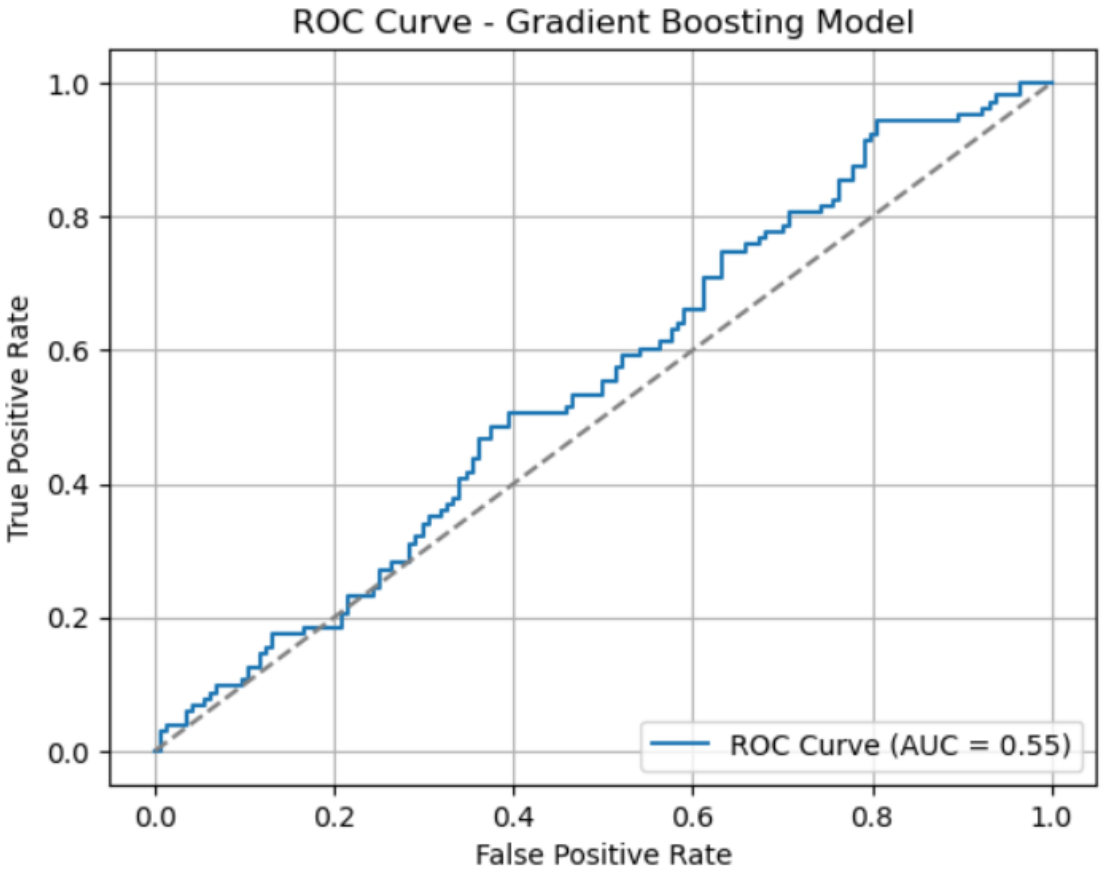
```
ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred)).plot()
```



We observe that the model predicts more false positives than negatives. However, since there are more negative cases, this is to be expected. The fraction of correctly classified positive instances is, in fact, larger than the fraction of incorrectly classified positives. This can be seen in more detail in the ROC curve.

```
y_proba = gbr.predict_proba(X_test)[:, 1]

fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Gradient Boosting Model")
```

```
plt.legend(loc="lower right")
plt.grid()
plt.show()
```



The curve shows that the classification is slightly better than chance, with the true positive rate being larger than the false positive rater for many possible thresholds, matching our previous observation. We can summarize the results of the model using a `classification_report`.

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.59      0.73      0.65       144
           1       0.43      0.28      0.34       103

    accuracy                           0.54       247
   macro avg       0.51      0.51      0.49       247
weighted avg       0.52      0.54      0.52       247
```

Indeed, the performance for true positives leaves a lot to be desired. Such a low recall indicates that the majority of instances labelled as positive are false positives. Moreover, we can see in the confusion matrices that the majority of upwards movements are missed by the model.

C.2 Hyperparameter Tuning

Now that we have built the skeleton of the model and obtained a sense of its performance, we proceed to build a family of Gradient Boosting Classifiers with different hyperparameters. Our choices are

- `max_depth` = 2, 3, 4, 5, 6
- `learning_rate` = 0.001, 0.01, 0.1, 0.2, 0.3, 0.4
- `n_estimators` = 50, 100, 150, 200, 250

We can train all these model at once using `GridSearchCV` in `scikit-learn`. This will search for the hyperparameters that yield the best `recall` (specified under `scoring`). Additionally, we improve the model by adding time-series cross-validation.

```
tscv = TimeSeriesSplit(n_splits=5, gap=1)

param_grid = {'n_estimators': [50, 100, 150, 200, 250],
'learning_rate': [0.001, 0.01, 0.1, 0.2, 0.3, 0.4],
'max_depth': [2, 3, 4, 5, 6]}

gbc = GradientBoostingClassifier()

grid_search = GridSearchCV(estimator=gbc, param_grid=param_grid, cv=tscv,
scoring='recall', n_jobs=-1, verbose = 1)

grid_search.fit(X_train, y_train)
```

The best parameters are found via the following script.

```
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
print(best_params)
```

In our case, `n_estimators = 50`, `max_depth = 3`, and `learning_rate = 0.4`.

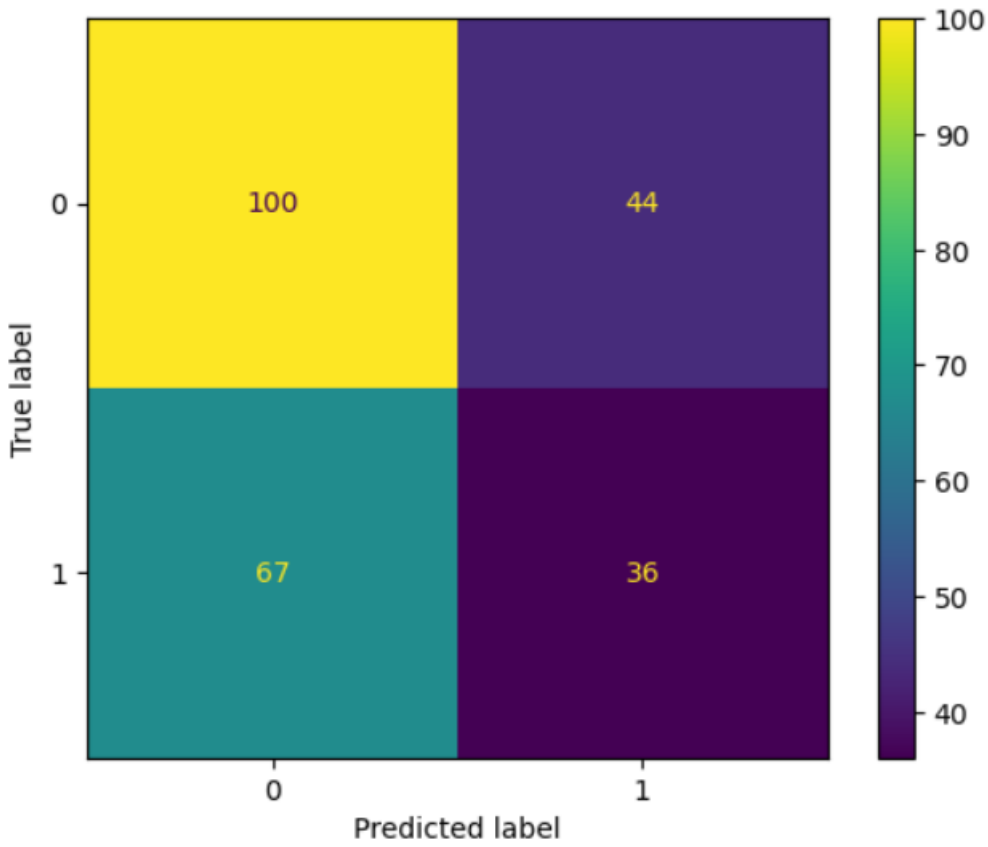
C.3 Model prediction quality

As in C.1, we can find the test recall score

```
y_pred_best = best_model.predict(X_test)
recall_best = recall_score(y_test, y_pred_best)
print(recall_best)
```

which, in this case, is 0.35, a substantial increase over 0.28 from the previous arbitrary choice. We can plot the confusion matrix,

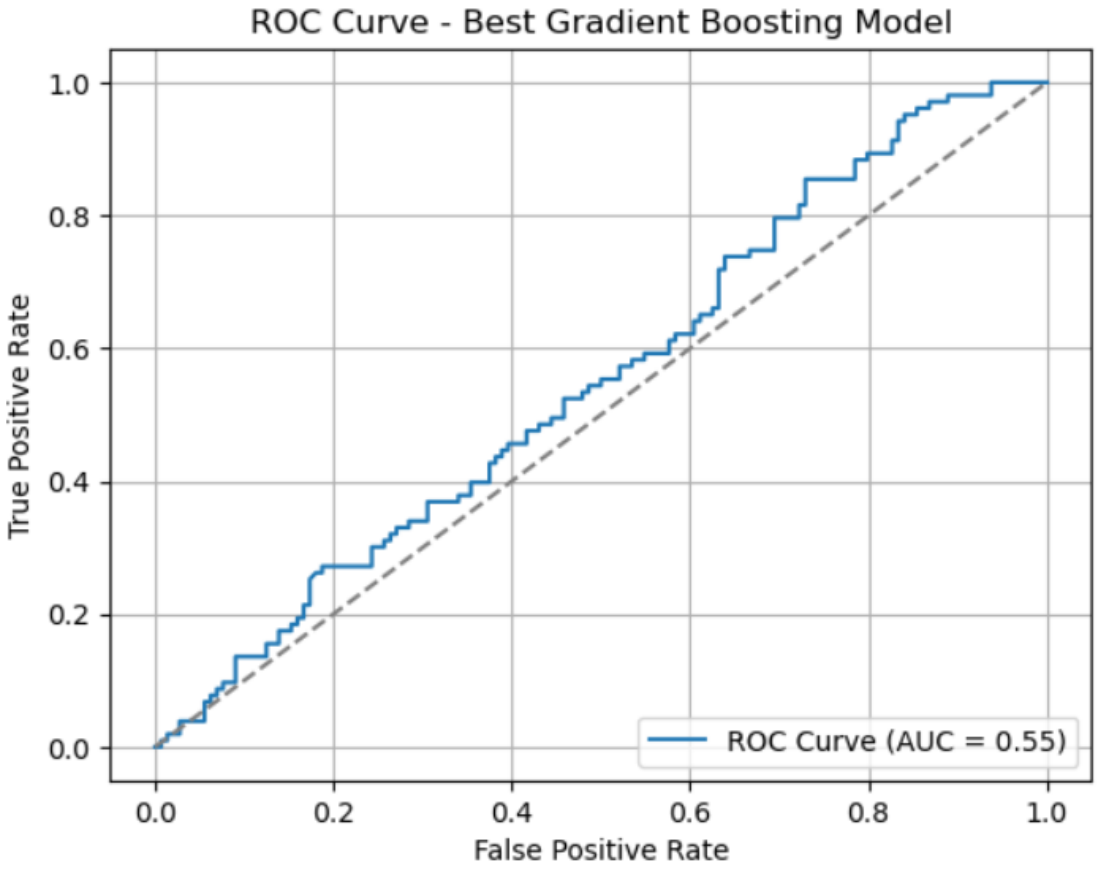
```
ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred_best)).plot()
```



which shows a decrease in incorrect classification and an increase in recall and overall true positive identification. However, it is still the case that most instances that are classified as an upwards movement are false positives, and most of the true uptrend is missed.

Let us now verify if there are any improvements in the ROC curve.

```
from sklearn.metrics import roc_curve, auc, RocCurveDisplay

y_proba = best_model.predict_proba(X_test)[:, 1]

fpr, tpr, thresholds = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve - Best Gradient Boosting Model")
plt.legend(loc="lower right")
plt.grid()
plt.show()
```



Unfortunately, the area under the ROC remains unchanged, meaning that the model does not perform much better even if we change thresholds. This is not surprising, however, since the hyperparameters of the best model do not differ much from the arbitrarily selected model.

The classification report gives us a summary of our observations

```
print(classification_report(y_test, y_pred_best))
```

```
              precision    recall  f1-score   support

           0       0.60      0.69      0.64       144
           1       0.45      0.35      0.39       103

    accuracy                           0.55       247
   macro avg       0.52      0.52      0.52       247
weighted avg       0.54      0.55      0.54       247
```

The larger recall is expected, since we used this indicator to score the models. The F1-score is also better for the optimal model for the positive case, making its overall predictive power superior.

C.4 Backtesting predicted signal with a trading strategy

Our observations show that, indeed, predicting movements of stock prices is a difficult task. Even though we tested a variety of models, they seem to perform only slightly better than chance. Even so, the results beg the question of whether one can use the slight edge to devise a trading strategy that can give investors an edge.

As mentioned before, we consider a long-term investor that buys a share when the model predicts the price will trend upwards the next day, and holds otherwise. We will compare this strategy with an investor that buys shares randomly, exposing themselves to different prices without information about the near-term behaviour of the stock.

To backtest, we have the model predict the entirety of the data and assign a new "Signal" column to the dataset. We also extract the price `latest_price` from the last available data point (May 9th 2025).

```
csu["Signal"] = best_model.predict(X3)
latest_price = csu["Close"].iloc[-1]
```

We assume that the price right before close is equal to the price at close, so that the investor is able to buy at the close price. We also assume that the trader buys exactly one share at close. We assign a columns `gains` to track the total increase in the price of the share up to the present time, and `investment`, which tracks the total amount of money invested.

```
csu["gains"] = (latest_price - csu["Signal"]*csu["Close"])*csu["Signal"]
csu["investment"] = csu["Signal"]*csu["Close"]
```

The total earning and investement can also be calculated.

```
total_gains = csu["gains"].sum()
total_investment = csu["investment"].sum()
percentage_return = (total_gains + total_investment)/total_investment
```

The percentage return is calculated to be 88% with this strategy.

We can simulate the strategy where the investor buys randomly. We will assume that the buy as many shares as the investor that follows the buy signals. To do this, we simulated Bernoulli trials with probability equal to the fraction of buy signals.

```
buy_signals = csu["Signal"].value_counts()[1]
hold_signals = csu["Signal"].value_counts()[0]
csu["Rand signal"] = np.random.binomial(1, buy_signals/(buy_signals+hold_signals), len(csu))
csu["random investment"] = csu["Rand signal"]*csu["Close"]
csu["random gains"] =  (latest_price - csu["Rand signal"]*csu["Close"])*csu["Rand signal"]

total_gains = csu["random gains"].sum()
total_investment = csu["random investment"].sum()

(total_gains + total_investment)/total_investment
```

Total earnings here are random depending on the dates of investment, but are observed to be around 90%. This means that the randommly behaving investor earns slightly more than the one based on the model. The reason for this is that the random investor does not wait until the prices start going up to decide on purchasing a share. Therefore, they might be exposed to lower prices as well.

The model, therefore, is not useful in a practical scenario as far as long-term long-position portfolios are concerned. A more profitable strategy would be to predict falls in the prices and buy the troughs. However, the model as presented only predicts uptrends, so such a scheme cannot be informed by it. Moreover, the model does not predict the magnitude of the price increases, which could be useful for strategies that invest more when larger jumps are anticipated.

Conclusion

In this project, we attempted to build a Gradient Boosting Classifier to predict upwards trends for the `CSU.TO` ticker. The model performs slightly better than chance, yielding a larger proportion of true positives than false positives. However, for a long term investor looking to buy before the prices go up, this model does not provide an optimal strategy.

We must note that this investment strategy may not be optimal either. More complex strategies involving buying and selling at the correct times, predicting lower prices to maximize long term gains, or allowing to take short positions might be able to leverage machine learning models akin to the one presented above. However, a more complex model is necessary to predict dips in prices, possibly allowing for `Buy`, `Hold` and `Sell` signals.

The model presented also has room for improvement. We considered an initial set of features that is easily accessible from readily available information, but more complex data, such as sentiment

analysis information or financial ratios, might prove to be more predictive. Additionally, the model could be further improved by performing a better treatement of the initial data, pruning outliers and performing more in-depth exploratory analysis. Not all hyperparameters were tuned either. More fine-grained parameters at the level of the tree nodes might also result in slight improvements to the model.

The results of this work, while slightly disappointing, are not unexpected. Predicting stock prices is known to be an open problem in finance, and the existing models are known to have limits. If it were possible to make predictions with such simple models, then many investors would be able to leverage them. Unfortunately, such arbitrage opportunities are short-lived, since the market tends to correct itself.