NATIONAL UNIVERSITY
of Computer & Emerging Sciences

Parallel and Distributed Computing
Semester Project

Presented by:

22i-1959 Ismail Hafeez

22i-1914 Talha Kayani

22i-1979 Abdul Rehman

Table of Contents

Project Overview

For our end-of-semester project, we were tasked with applying binary classification to a dataset to train the model and provide the following:

1. Accuracy
2. F1 score
3. Confusion matrix

This was just the first task. The main objective was to apply parallel and distributed frameworks to reduce the model training time by a minimum of 70 percent.

As far as the tech stack is concerned regarding time optimization, we were free to choose from the various frameworks available, so long as the objective was reached.

Tech stack

Language: Python
Libraries: Pandas, Numpy, Matplotlib, Seasborn, Scikit-Learn, time
Distributed System: Dask
IDE: Visual Studio Code
OS: Linux (Ubuntu 24.04.2 LTS)

Data Pre-processing

Please refer to this snapshot of the dataset provided. Shape = (41000, 8)

	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	target
0	27.75	55.0	B	875.98	No	8	44.16	0
1	31.33	39.0	C	839.91	No	4	13.93	1
2	23.87	10.0	A	1364.70	Yes	4	15.16	0
3	35.64	34.0	C	1462.07	No	4	15.71	1
4	NaN	NaN	C	710.61	No	8	13.77	1

Handling Missing Values

There were quite many missing values, as we can see from the snapshot below. To cater this, we filled in the missing values using the mean method. This ensured that our missing values were catered in the best way possible.

```
feature_1    2054
feature_2    2050
feature_3       0
feature_4    2054
feature_5       0
feature_6       0
feature_7    2036
target          0
dtype: int64
```

```
num_imputer = SimpleImputer(strategy="mean")
df[['feature_1', 'feature_2', 'feature_4',
'feature_7']] =
num_imputer.fit_transform(df[['feature_1',
'feature_2', 'feature_4', 'feature_7']])
```

Encoding Categorical Values

Columns ‘feature_3’ and ‘feature_5’ contained categorical values (‘A’, ‘B’, ‘C’ and ‘Yes’,’No’ respectively). To ensure the data type of our dataframe is uniform, we had to turn these values into numerical values, like so.

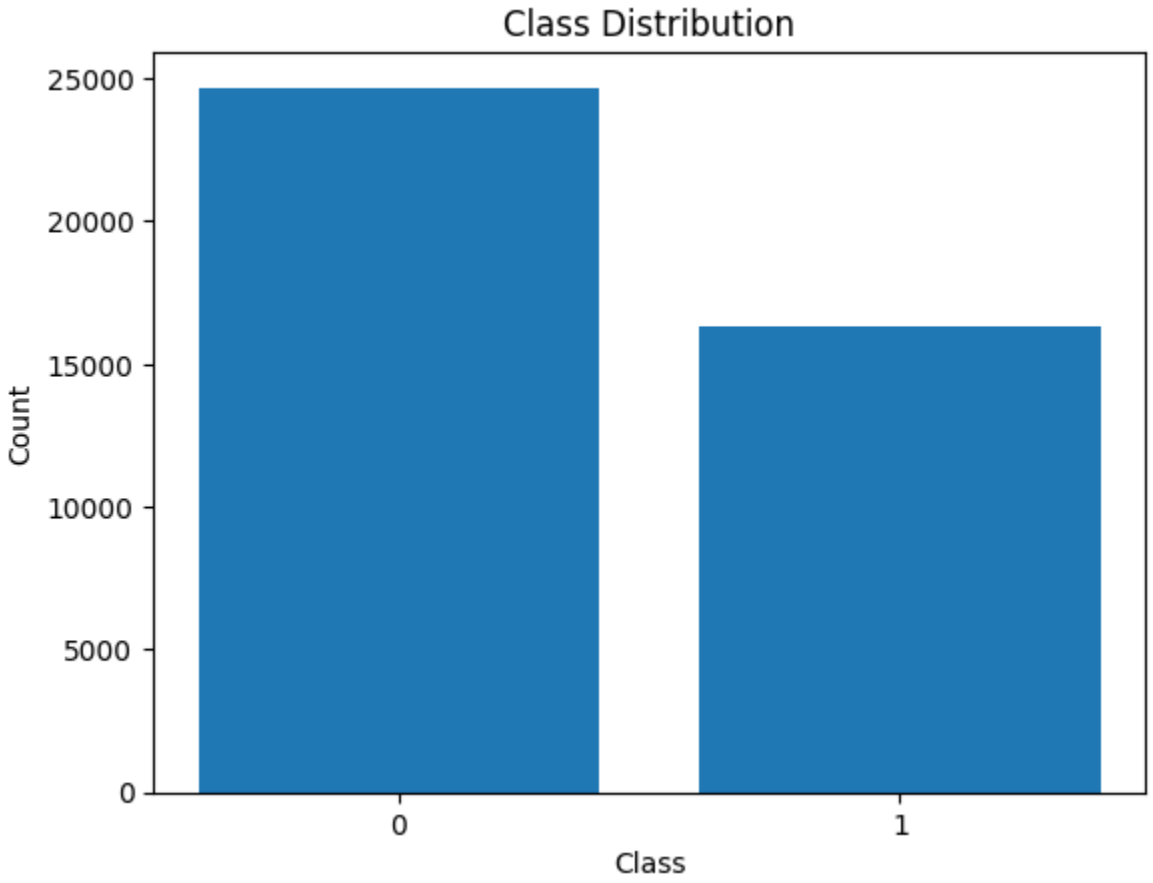
```
# feature_3 (A/B/C) → Use Label Encoding (A=0, B=1, C=2).
le = LabelEncoder()
df['feature_3'] = le.fit_transform(df['feature_3'])

# feature_5 (Yes/No) → Map manually (Yes=1, No=0).
df['feature_5'] = df['feature_5'].map({'Yes': 1, 'No': 0})
```

Problems with Data

Like any dataset, our data wasn't without any headaches.

Class Distribution

The data's target column 'target' contains two classes. 0 and 1. However, the distribution of these classes is somewhat imbalanced.



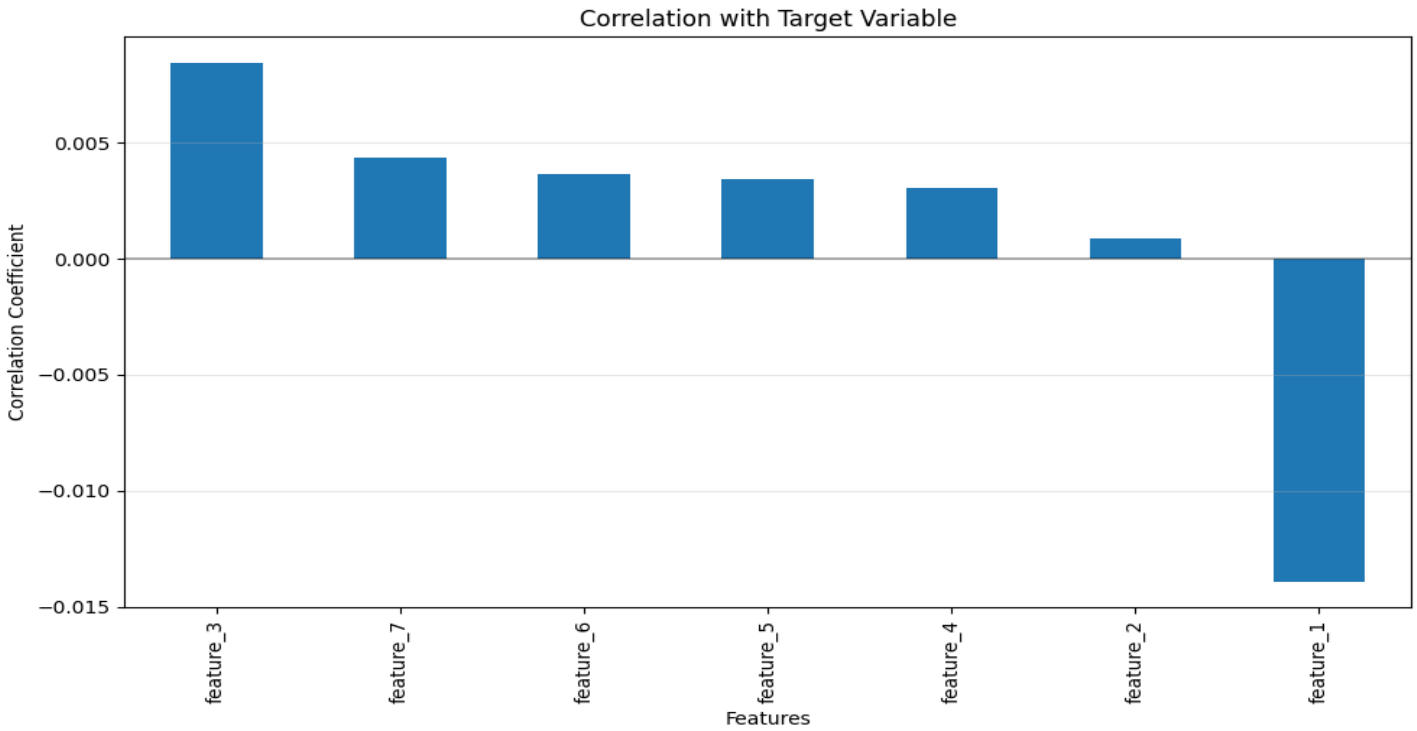
As we can see from the chart above, the class distribution is split on a 3:2 ratio. This highly affected the model's F1 score, as discussed later on.

Correlation

Our next step was to try and figure out the correlation between the features and the target. In other words, we want to know which column influences the target values, as that would help us later on in training our model and choosing the right parameters.

	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	target
feature_1	1.000000	-0.005120	-0.001790	-0.002596	-0.001128	-0.006288	-0.010552	-0.013926
feature_2	-0.005120	1.000000	-0.002838	-0.002840	-0.004140	-0.000947	0.007323	0.000890
feature_3	-0.001790	-0.002838	1.000000	-0.007419	0.015162	0.008983	0.008237	0.008448
feature_4	-0.002596	-0.002840	-0.007419	1.000000	-0.001626	0.007698	-0.003203	0.003063
feature_5	-0.001128	-0.004140	0.015162	-0.001626	1.000000	0.004775	-0.000590	0.003432
feature_6	-0.006288	-0.000947	0.008983	0.007698	0.004775	1.000000	-0.005197	0.003625
feature_7	-0.010552	0.007323	0.008237	-0.003203	-0.000590	-0.005197	1.000000	0.004343
target	-0.013926	0.000890	0.008448	0.003063	0.003432	0.003625	0.004343	1.000000

Unfortunately for us, the correlation was a disappointment. As we can see from the correlation matrix above, and the chart below, none of the features have a strong correlation with the target.



Now, because columns ‘feature_1’ and ‘feature_2’ have an extremely low correlation with the target, one would assume that the next step would surely be to remove these columns as they influence virtually nothing towards the target, but that would be wrong.

All correlations are extremely weak - The strongest correlation is only about 0.007, which is effectively negligible. In practical terms, none of the features show meaningful linear correlation with the target. Correlation only captures linear relationships - Low correlation doesn't mean a feature is useless. It only means there isn't a strong linear relationship. The features might have:

1. Non-linear relationships with the target
2. Interaction effects with other features
3. Conditional relationships that only appear in subsets of the data

Hence, we try to find other forms of non-linear correlation to compute feature importance. One such method is called the Decision Tree Classifier method.

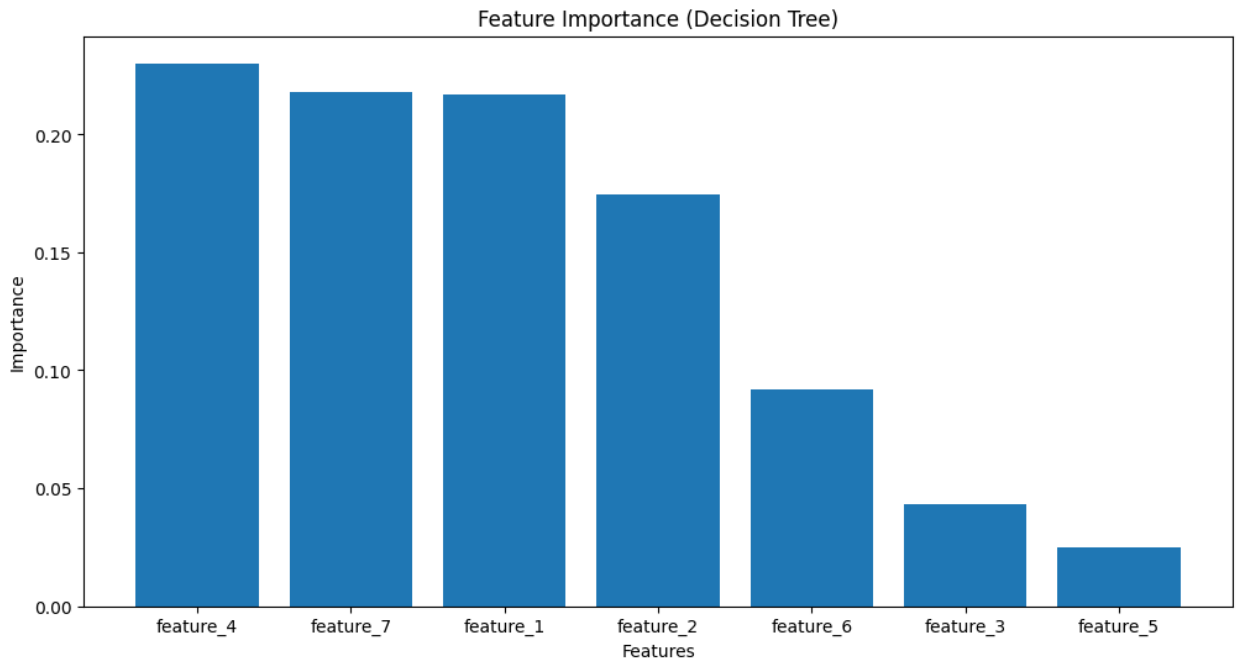
Decision trees determine feature importance by:

- Calculating how much each feature split reduces impurity in the data
- Weighting this reduction by the proportion of samples reaching that node
- Summing these weighted reductions across all nodes where the feature is used

This approach captures both linear and non-linear relationships between features and the target, which is why it can be more insightful than correlation for complex relationships.

```
X = df.drop('target', axis=1)
y = df['target']

clf = DecisionTreeClassifier(random_state=0)
clf.fit(X, y)

tree_importances = pd.DataFrame({
   'Feature': X.columns,
   'Importance': clf.feature_importances_
}).sort_values(by='Importance', ascending=False)

print(tree_importances)

X_bar = tree_importances['Feature']
Y_bar = tree_importances['Importance']


plt.figure(figsize=(12, 6))
```

```
plt.bar(X_bar, Y_bar)
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importance (Decision Tree)')
plt.show()
```



As we can see from the chart above, dropping features 1 and 2 would not have been beneficial, because they possess a strong non-linear relationship with the target.

Feature Engineering

Due to the low correlation of existing features with the target, our next next hope lies in feature engineering - a process where we create new features from existing features by computing some numericals.

```
# Step 3: Interaction Features
df['f1_by_f2'] = df['feature_1'] / (df['feature_2'] + 1e-5)  # avoid
division by zero
df['f4_by_f6'] = df['feature_4'] / (df['feature_6'] + 1e-5)

# Step 4: Binning Feature_7
df['feature_7_bin'] = pd.qcut(df['feature_7'], q=3, labels=['low',
'medium', 'high'])
df = pd.get_dummies(df, columns=['feature_7_bin'])

# Step 5: Domain-Inspired Feature
df['cost_per_unit'] = df['feature_4'] / (df['feature_6'] + 1e-5)

# Step 6: Polynomial Features (on numerical only)
poly_features = ['feature_1', 'feature_2']
poly = PolynomialFeatures(degree=2, interaction_only=True,
include_bias=False)
poly_df = pd.DataFrame(poly.fit_transform(df[poly_features]),
columns=poly.get_feature_names_out(poly_features))
df = pd.concat([df, poly_df.drop(columns=poly_features)], axis=1)
```

With this, we conclude our data pre-processing. The manipulated data is now saved as csv for model training.

```
# Destination Folder
dsc_folder = '/home/ismail/Desktop/PDC_Project/data/'
df.to_csv(f'{dsc_folder}cleaned_data.csv',index=False)
```

Sequential

XGBoost

```
start = time.time()

# 3. Split features and target
X = df.drop('target', axis=1)
y = df['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 4. Train XGBoost model
model = XGBClassifier(
    use_label_encoder=False,
    eval_metric='logloss',
    n_estimators=900,
    max_depth=11,
    learning_rate=0.05,
    subsample=0.9,
    colsample_bytree=0.9,
    scale_pos_weight=1.5,
    random_state=42
)
model.fit(X_train, y_train)

# 5. Make predictions
y_pred = model.predict(X_test)

# 6. Evaluate
acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

end = time.time()

print(f"Accuracy: {acc:.4f}")
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
```

```
print(cm)
print(f"\nTotal Processing Time: {end - start:.2f} seconds")
```

```
=== Output ===

Accuracy: 0.5404
F1 Score: 0.4134
Confusion Matrix:
[[3103 1824]
 [1945 1328]]

Total Processing Time: 38.42 seconds
```

Random Forest

```
start = time.time()

# Final Split - Separate features and target:
X = df.drop('target', axis=1)
y = df['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 4. Handle imbalance using SMOTE
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_resample(X_train, y_train)

# 5. Train Random Forest model
model = RandomForestClassifier(
   class_weight="balanced",
   n_estimators=100,
   random_state=42
)
model.fit(X_train, y_train)

# 6. Make predictions
y_pred = model.predict(X_test)

# 7. Evaluate
```

```
acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

end = time.time()
seq_time = end - start

print(f"Accuracy: {acc:.4f}")
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
print(cm)
print(f"\nTotal Processing Time: {seq_time:.2f} seconds")
```

=== Output ===

```
Accuracy: 0.5898
F1 Score: 0.3078
Confusion Matrix:
[[4088  839]
 [2525  748]]

Total Processing Time: 17.97 seconds
```

Parallel

For reducing time, we turned to Dask - an open-source Python library for parallel computing.
Dask scales Python code from multi-core local machines to large distributed clusters in the cloud. It provides a familiar user interface by mirroring the APIs of other libraries in the PyData ecosystem including: Pandas, scikit-learn and NumPy.
It also exposes low-level APIs that help programmers run custom algorithms in parallel.

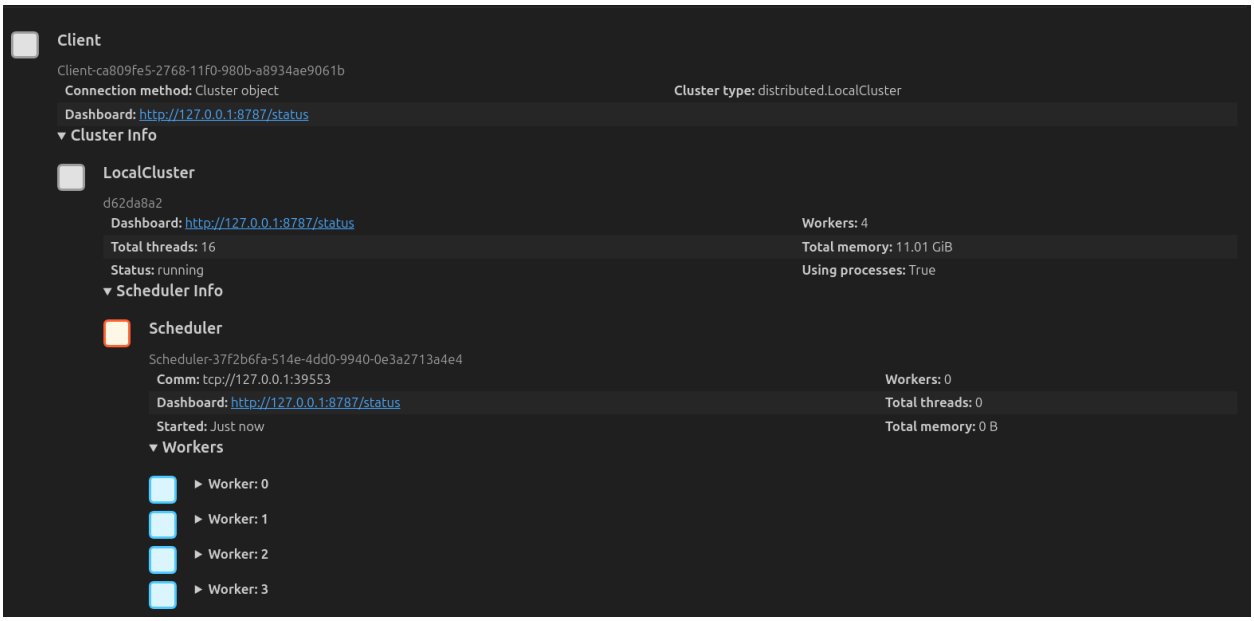
```
# Dask
import dask.dataframe as dd
from dask.distributed import Client, LocalCluster
from dask_ml.model_selection import train_test_split
from dask_ml.wrappers import ParallelPostFit
```

For utilizing Dask, firstly, we initialize the client and local cluster like so.

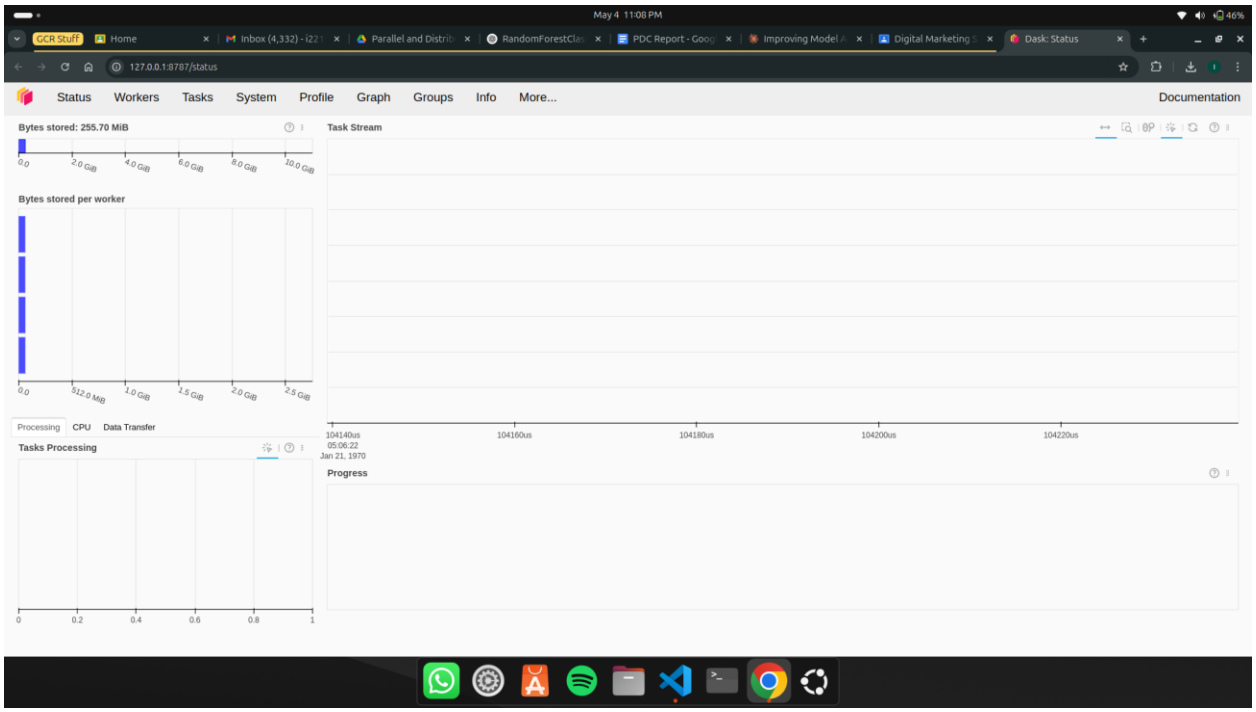
```
cluster = LocalCluster()
client = Client(cluster)
```

We can include the amount of cores and threads per core we need by giving certain parameters. However, by giving no parameters, Dask automatically detects the computer architecture in terms of cores, and uses the machine optimally.

We can see that the client has initiated four worker nodes.



Dask also has an integrated UI dashboard that can be viewed on localhost:`8787`.



Random Forest

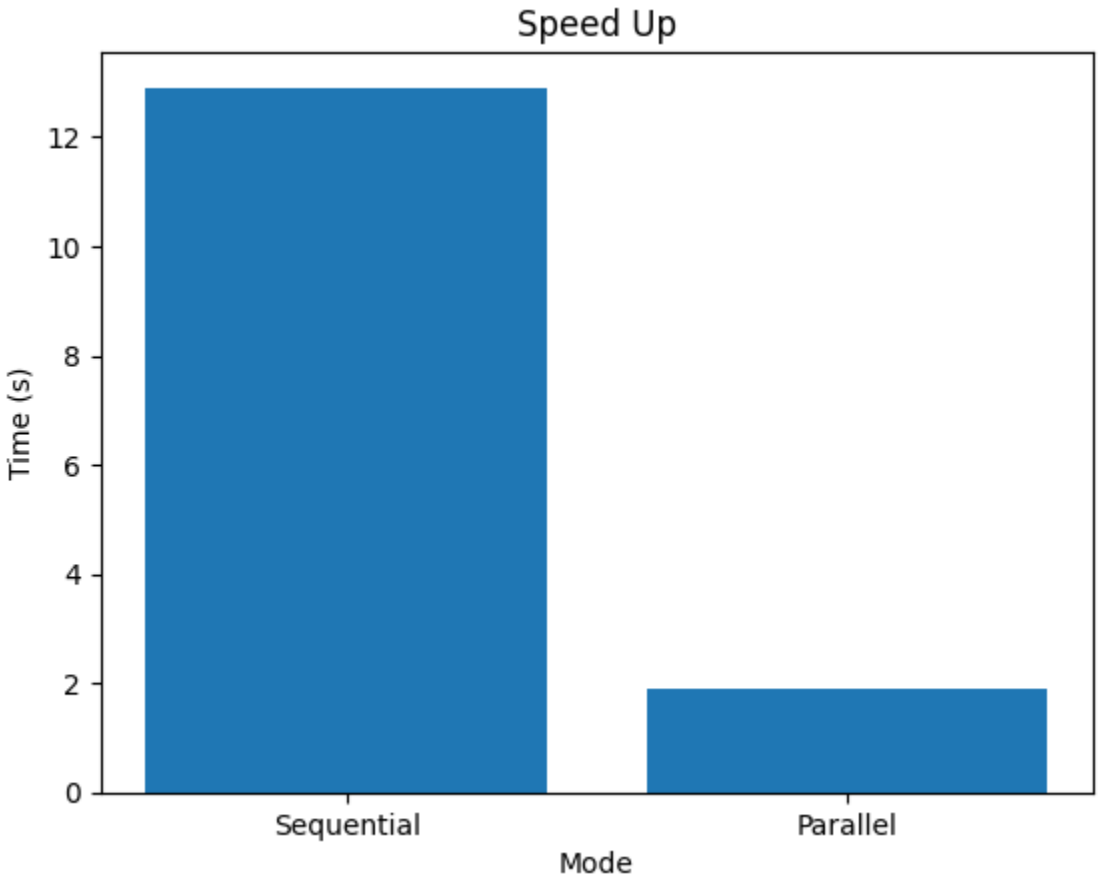
```
# Start local Dask cluster
cluster = LocalCluster()
client = Client(cluster)
print(f"Dashboard link: {client.dashboard_link}")

start = time.time()

# Final Split - Separate features and target:
X = pdf.drop('target', axis=1)
y = pdf['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, shuffle=True)

# Train Random Forest model using Dask's ParallelPostFit wrapper
model = ParallelPostFit(estimator=RandomForestClassifier(
   class_weight="balanced",
```

```
    n_estimators=100,
    random_state=42,
    n_jobs=-1)
)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate
acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

end = time.time()
par_time = end - start

print("===== Dask + Random Forest Performance =====")
print(f"Accuracy: {acc:.4f}")
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
print(cm)
print(f"\nTotal Processing Time: {par_time:.2f} seconds")
```

=== Output ===

```
Dashboard link: http://127.0.0.1:42629/status
===== Dask + Random Forest Performance =====
Accuracy: 0.5898
F1 Score: 0.3078
Confusion Matrix:
[[4088  839]
 [2525  748]]

Total Processing Time: 2.14 seconds
```

As we can see, we achieved a whopping 88% speed-up. However, this `percentage` often fluctuates, especially when trying different parameters and re-running the code over and over again.


Speed Up
Time (s)
0
2
4
6
8
10
12
Sequential
Parallel
Mode