Configuration Manual

MSc Research Project
Masters in Data Analytics

Kesav Swaroop Reddy Devarapati
Student ID: x23196459

School of Computing
National College of Ireland

Supervisor: Cristina Hava Muntean

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Kesav Swaroop Reddy Devarapati
Student ID:	x23196459
Programme:	Masters in Data Analytics
Year:	2024
Module:	MSc Research Project
Supervisor:	Cristina Hava Muntean
Submission Due Date:	29/01/2025
Project Title:	Configuration Manual
Word Count:	Approximately 1500 words
Page Count:	10

Signature:	D.Kesav Reddy
Date:	11th December 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Kesav Swaroop Reddy Devarapati
x23196459

1 Introduction

The goal of this project is to predict multiple biological targets from experimental gene expression and cell viability data. The machine learning problem is to implement Logistic Regression, Random Forest, XGBoost, LightGBM, CatBoost, Gradient Boosting, AdaBoost, K-Nearest Neighbors (KNN), and Decision Tree classifiers, in that order. The project directory consists of :
Research-Project-Code.ipynb : All the code for preprocessing, training and evaluation in a single Jupyter notebook.
Datasets : train-features.csv, train-targets-scored.csv, train-targets-nonscored.csv, test-features.csv

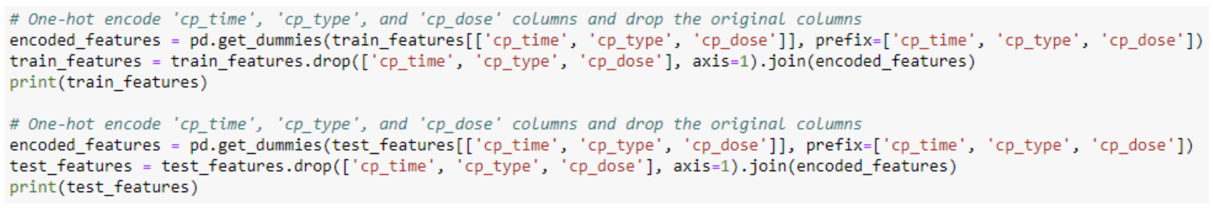
2 System Configuration

2.1 Hardware Requirements

Processor: Hence minimum quad core CPU (Intel i5 or equivalent AMD Ryzen 5).
Memory: Minimum 8 GB RAM (16 GB is recommended for a smooth running).
Disk Space: Datasets and intermediate outputs, and models at least 10 GB free.
GPU (Optional): Use a NVIDIA GPU specifically with CUDA support for accelerated processing, which is great with XGBoost, LightGBM, etc.

2.2 Software Requirements

Python: The entire development was being conducted in Python 3.8 to avoid any issues with developing while also using all the required libraries and frameworks. Keep things consistent with other project dependencies by making sure your system has Python 3.8 or higher installed.
Libraries: Pandas for Data manipulation and ingestion. High performance numerical computation using numpy. Moreover, matplotlib and seaborn for more advanced data visualisation. machine learning algorithms can be implemented using scikit learn for that. Other gradient boosting frameworks that work great with tabular data include, xgboost, lightgbm and catboost. Supplying progress bars as part of loops and processes makes things much more traceable while long running.



```python
# One-hot encode 'cp_time', 'cp_type', and 'cp_dose' columns and drop the original columns
encoded_features = pd.get_dummies(train_features[['cp_time', 'cp_type', 'cp_dose']], prefix=['cp_time', 'cp_type', 'cp_dose'])
train_features = train_features.drop(['cp_time', 'cp_type', 'cp_dose'], axis=1).join(encoded_features)
print(train_features)

# One-hot encode 'cp_time', 'cp_type', and 'cp_dose' columns and drop the original columns
encoded_features = pd.get_dummies(test_features[['cp_time', 'cp_type', 'cp_dose']], prefix=['cp_time', 'cp_type', 'cp_dose'])
test_features = test_features.drop(['cp_time', 'cp_type', 'cp_dose'], axis=1).join(encoded_features)
print(test_features)
```

Figure 1: Feature Encoding

3 Project Development

3.1 Dataset Overview

The datasets are primary to the project's analytical capacity. They include:
train-features.csv: It consists of 875 features indicating gene expression and cell viability under many conditions.
train-targets-scored.csv: Projects core predictive tasks comprise binary targets indicating the presence of biological activity.
train-targets-nonscored.csv: It also gives us additional unscored biological targets for supplementary analysis or feature engineering.
test-features.csv: These are features associated to the test set, used to assess how our model will generalise on the unseen data set.
train-drug.csv: The file can be used for stratified analyses, and for showing treatment specific responses.

3.2 Data Pre-processing

Preprocessing is essential to prepare the data for machine learning:

3.2.1 Feature Encoding

One hot encode the categorical variables (cp-time, cp-type, cp-dose) to create numerical representations that can be used by machine learning algorithms. The latter is then merged with remaining original dataset and original categorical columns are dropped.

3.2.2 Feature Scaling

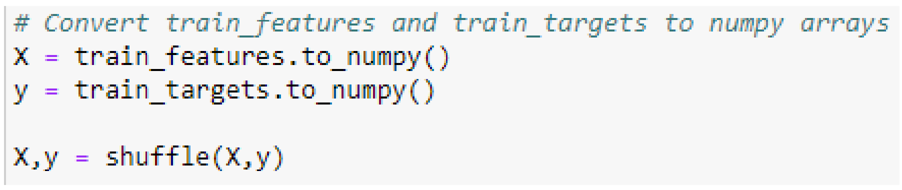
StandardScaler is used to scale all numerical features so that they are of same scale.

3.2.3 Splitting Features and Targets

The target labels (train-targets-scored.csv) are separated from feature columns for the training of the model. Both features and targets are purged of the sig-id column to guarantee data integrity.

3.2.4 Data Shuffling

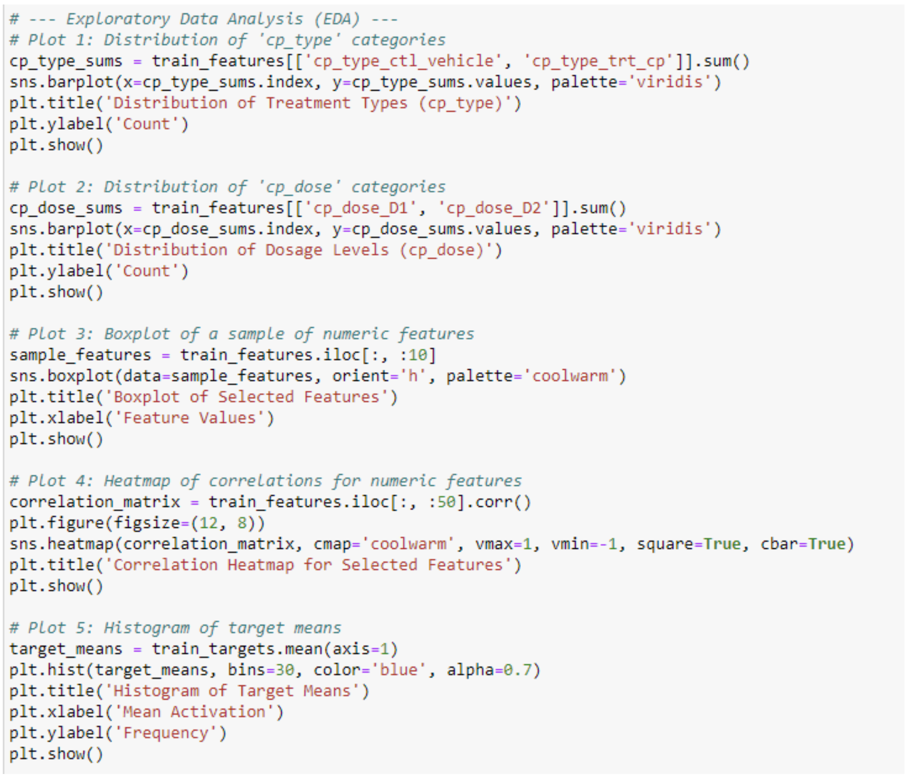
To avoid biases during the model training, data is shuffled.



```python
# Convert train_features and train_targets to numpy arrays
X = train_features.to_numpy()
y = train_targets.to_numpy()

X,y = shuffle(X,y)
```

Figure 2: Data Shuffling



```python
# --- Exploratory Data Analysis (EDA) ---
# Plot 1: Distribution of 'cp_type' categories
cp_type_sums = train_features[['cp_type_ctl_vehicle', 'cp_type_trt_cp']].sum()
sns.barplot(x=cp_type_sums.index, y=cp_type_sums.values, palette='viridis')
plt.title('Distribution of Treatment Types (cp_type)')
plt.ylabel('Count')
plt.show()

# Plot 2: Distribution of 'cp_dose' categories
cp_dose_sums = train_features[['cp_dose_D1', 'cp_dose_D2']].sum()
sns.barplot(x=cp_dose_sums.index, y=cp_dose_sums.values, palette='viridis')
plt.title('Distribution of Dosage Levels (cp_dose)')
plt.ylabel('Count')
plt.show()

# Plot 3: Boxplot of a sample of numeric features
sample_features = train_features.iloc[:, :10]
sns.boxplot(data=sample_features, orient='h', palette='coolwarm')
plt.title('Boxplot of Selected Features')
plt.xlabel('Feature Values')
plt.show()

# Plot 4: Heatmap of correlations for numeric features
correlation_matrix = train_features.iloc[:, :50].corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, cmap='coolwarm', vmax=1, vmin=-1, square=True, cbar=True)
plt.title('Correlation Heatmap for Selected Features')
plt.show()

# Plot 5: Histogram of target means
target_means = train_targets.mean(axis=1)
plt.hist(target_means, bins=30, color='blue', alpha=0.7)
plt.title('Histogram of Target Means')
plt.xlabel('Mean Activation')
plt.ylabel('Frequency')
plt.show()
```

Figure 3: Exploratory Data Analysis

3.3 Exploratory Data Analysis

EDA provides insights into the dataset:

3.3.1 Distributions

We use bar plots to visualize class distributions of treatment types (cp-type) and dosage levels (cp-dose).

3.3.2 Correlation Heatmaps

In order to identify redundancy and multicollinearity, correlations between features are visualized.

3.3.3 Feature Distributions

Numerical properties are studied with box plots and density plots, and the observations of numerical features are summarized through box and whiskers plots.

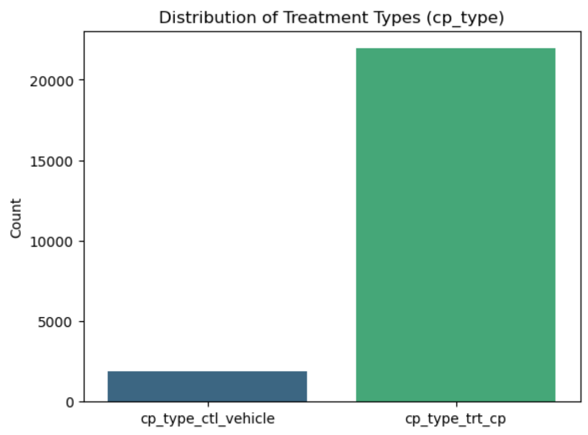


Figure 4: Distribution of Treatment Types

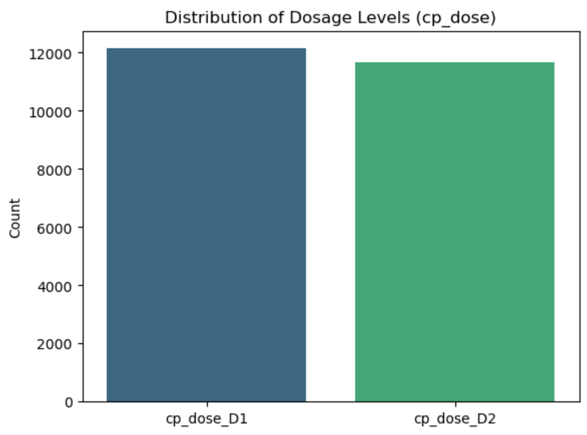


Figure 5: Distribution of Dosage Levels

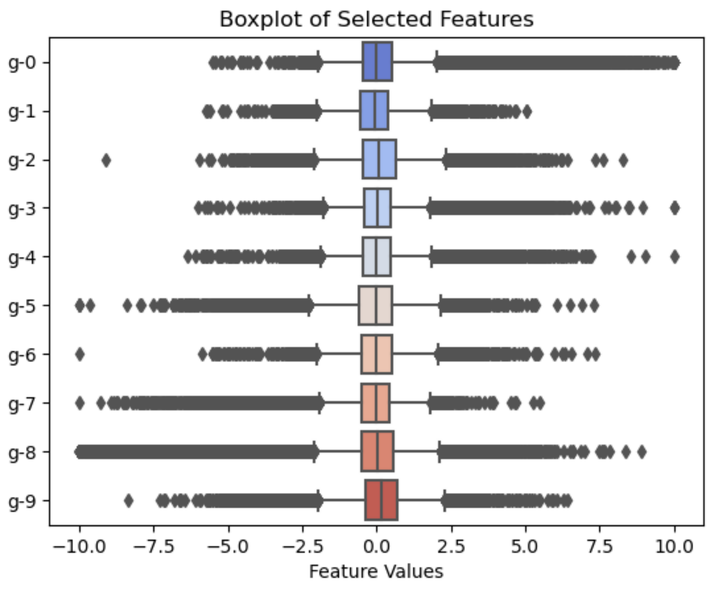


Figure 6: Box plot for selected features

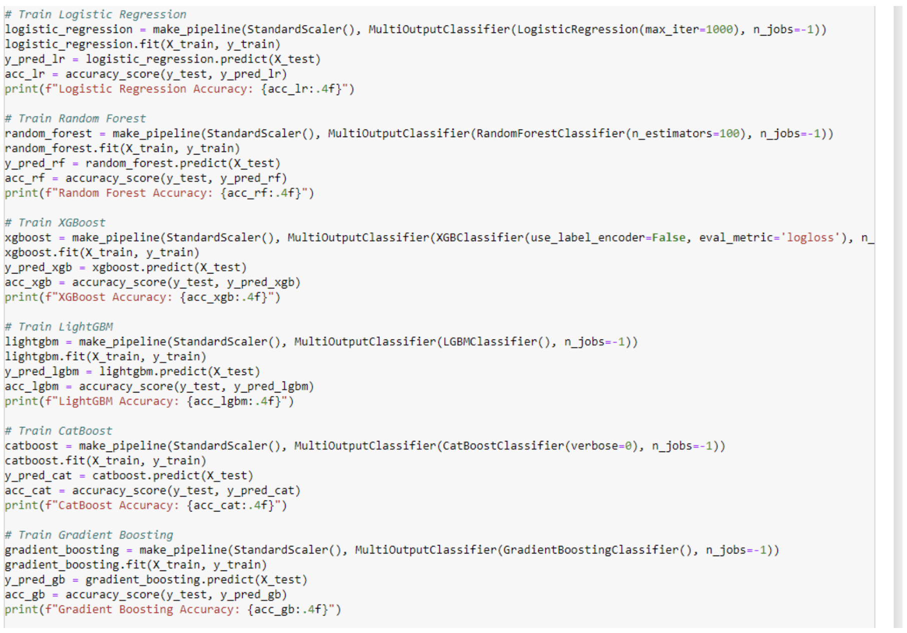
3.4 Model Selection

3.4.1 Logistic Regression

Often as a baseline linear classifier for binary and multi label classification tasks, Logistic Regression is a method. It supports the probability of a given class belonging to certain label with the use of a sigmoid function. It is remarkably simple, and is very effective in linearly separable datasets. Yet without feature transformation or engineering, it has trouble with any non-linear relationship.

3.4.2 Random Forest

Bagging ensemble model includes Random Forest, here it will build many Decision Trees by using different subsets of data and get together their predictions. Using this approach we reduce overfitting and increase the generalizability, making this approach robust for multi-label tasks. The problem that Random Forest excels at is high-dimensional data and mixed feature types.



```
# Train Logistic Regression
logistic_regression = make_pipeline(StandardScaler(), MultiOutputClassifier(LogisticRegression(max_iter=1000), n_jobs=-1))
logistic_regression.fit(X_train, y_train)
y_pred_lr = logistic_regression.predict(X_test)
acc_lr = accuracy_score(y_test, y_pred_lr)
print(f"Logistic Regression Accuracy: {acc_lr:.4f}")

# Train Random Forest
random_forest = make_pipeline(StandardScaler(), MultiOutputClassifier(RandomForestClassifier(n_estimators=100), n_jobs=-1))
random_forest.fit(X_train, y_train)
y_pred_rf = random_forest.predict(X_test)
acc_rf = accuracy_score(y_test, y_pred_rf)
print(f"Random Forest Accuracy: {acc_rf:.4f}")

# Train XGBoost
xgboost = make_pipeline(StandardScaler(), MultiOutputClassifier(XGBClassifier(use_label_encoder=False, eval_metric='logloss'), n_
xgboost.fit(X_train, y_train)
y_pred_xgb = xgboost.predict(X_test)
acc_xgb = accuracy_score(y_test, y_pred_xgb)
print(f"XGBoost Accuracy: {acc_xgb:.4f}")

# Train LightGBM
lightgbm = make_pipeline(StandardScaler(), MultiOutputClassifier(LGBMClassifier(), n_jobs=-1))
lightgbm.fit(X_train, y_train)
y_pred_lgbm = lightgbm.predict(X_test)
acc_lgbm = accuracy_score(y_test, y_pred_lgbm)
print(f"LightGBM Accuracy: {acc_lgbm:.4f}")

# Train CatBoost
catboost = make_pipeline(StandardScaler(), MultiOutputClassifier(CatBoostClassifier(verbose=0), n_jobs=-1))
catboost.fit(X_train, y_train)
y_pred_cat = catboost.predict(X_test)
acc_cat = accuracy_score(y_test, y_pred_cat)
print(f"CatBoost Accuracy: {acc_cat:.4f}")

# Train Gradient Boosting
gradient_boosting = make_pipeline(StandardScaler(), MultiOutputClassifier(GradientBoostingClassifier(), n_jobs=-1))
gradient_boosting.fit(X_train, y_train)
y_pred_gb = gradient_boosting.predict(X_test)
acc_gb = accuracy_score(y_test, y_pred_gb)
print(f"Gradient Boosting Accuracy: {acc_gb:.4f}")
```

Figure 7: Training the models

3.4.3 XGBoost

An implementation of gradient boosted decision trees: Extreme Gradient Boosting (XGBoost). It is a speed and performance optimized tool which uses a variety of regularization techniques for reducing the over fit. XGBoost is very useful for structured dataset, gives higher accuracy and also ignore missing values.

3.4.4 LightGBM

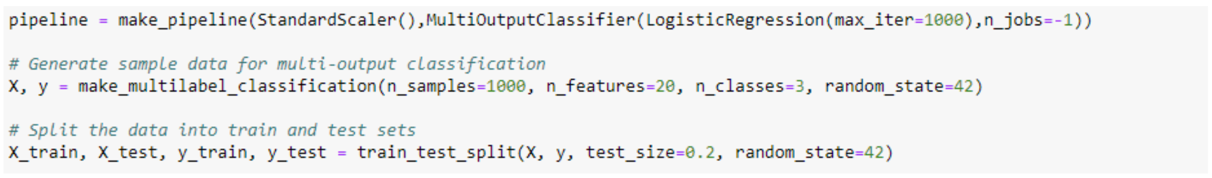
Gradient boosting framework, lightGBM (Light Gradient Boosting Machine) is another speed and efficiency focused framework. For large number of datapoints, with high feature dimensionality, it handles it efficiently using a histogram based algorithm. Being very scalable and also able to handle categorical features directly, LightGBM is obviously very good for solving multi label problems.

3.4.5 CatBoost

A gradient boosting algorithm tuned for categorical data, catboost. It can handle categorical variables natively without need for extensive preprocessing; unlike other gradient boosting models. It exhibits robust performance reducing overfitting and high interpretability for feature importance.

3.4.6 Gradient Boosting

In Gradient Boosting, we have a sequential ensemble technique which builds up models in an incremental fashion to reduce errors made by previous models. However, the flexibility of their loss functions means that each model is minimizing a differentiable loss function. It is slower than bagging models like Random Forest, but often has higher accuracy because it emphasizes the hard to predict data points.



```
pipeline = make_pipeline(StandardScaler(),MultiOutputClassifier(LogisticRegression(max_iter=1000),n_jobs=-1))

# Generate sample data for multi-output classification
X, y = make_multilabel_classification(n_samples=1000, n_features=20, n_classes=3, random_state=42)

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 8: Pipeline

3.4.7 AdaBoost

Adaptive Boosting (AdaBoost) is an ensemble method, where weak learners, usually just decision stumps, are combined to form a powerful predictive model. Then it has the effect of assigning higher weights for misclassified examples, which will cause subsequent models toinstruction, which models will pay more attention to these errors. For moderately imbalanced datasets, we find that AdaBoost is effective, but it can be sensitive to noisy data.

3.4.8 K-Nearest Neighbors (KNN)

KNN is an instance based non parametric learning algorithm to predict labels from the labels of the nearest neighbors. The payoff is simple to implement and interpret but computationally intensive with large datasets. However, KNN does well for smaller datasets with well defined class distribution.

3.4.9 Decision Tree

Then Decision Tree is a tree based algorithm that makes prediction by recursively splitting data based on feature values. It is easy to speak, understand and visualize it for exploratory analysis. Decision Trees do overfit however and this can be mitigated by pruning or combining together multiple trees in ensemble methods.

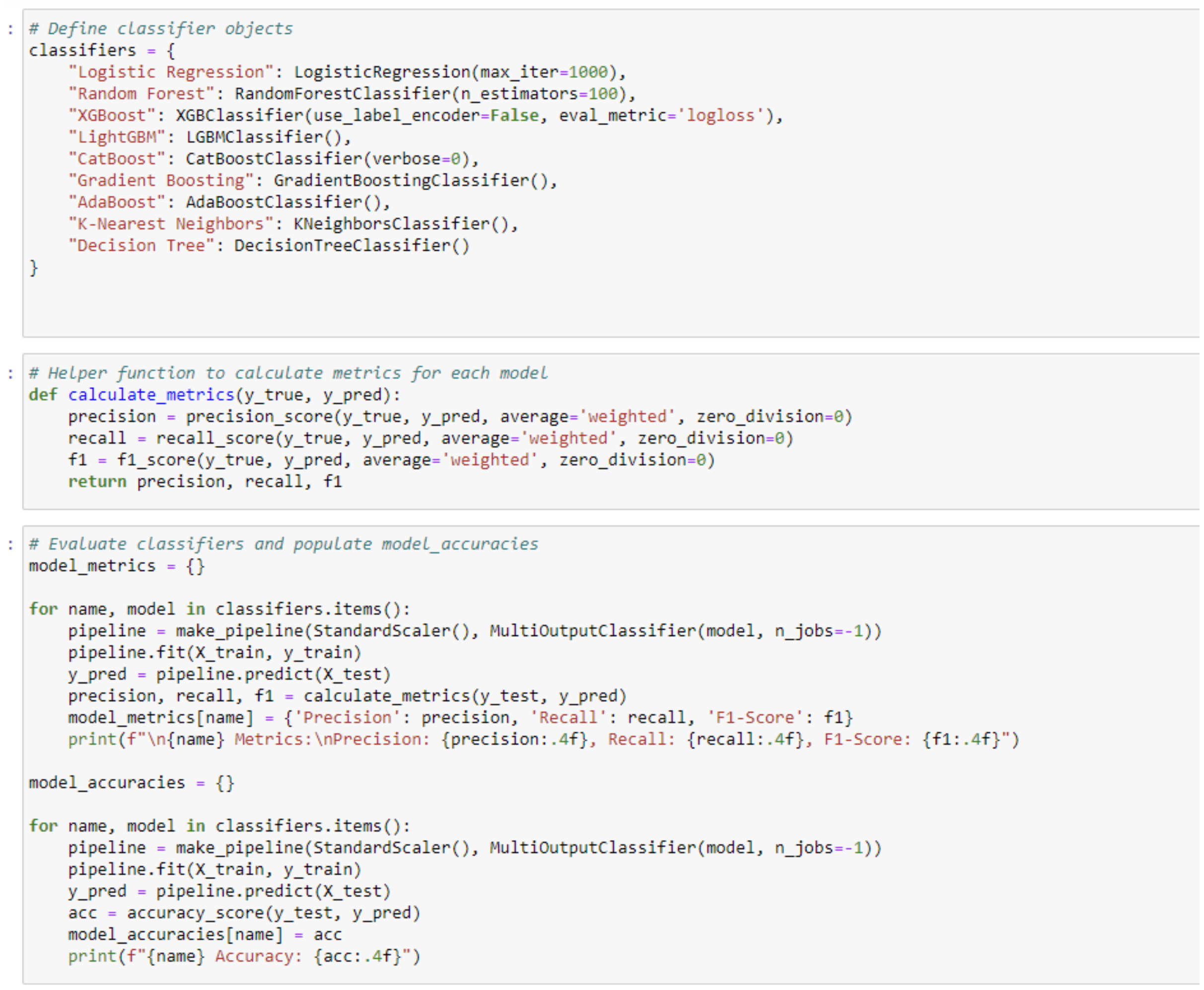
3.5 Model Training

3.5.1 Pipeline Creation

Each machine learning model was wrapped inside of a pipeline to streamline the training and evaluation process. The pipeline included a StandardScaler to normalize the features and a MultiOutputClassifier to deal with the multi label shape of the classification problem. The pipeline scaled the data so that the scales of different features did not bias the models. Each base model was wrapped to the MultiOutputClassifier which could predict multiple target labels at once. This also allowed for modularity in which we were able to easily swap out models or additional preprocessing steps.

3.6 Model Evaluation

To comprehensively assess model performance, several evaluation metrics were employed:
Accuracy: This is a metrics that computes the percentage of the predicted labeled correct to total labeled, aggregating the model performance. However, accuracy is not always accurate when imbalanced.
Precision, Recall, and F1-Score: A more nuanced view of model performance was



```
: # Define classifier objects
classifiers = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=100),
    "XGBoost": XGBClassifier(use_label_encoder=False, eval_metric='logloss'),
    "LightGBM": LGBMClassifier(),
    "CatBoost": CatBoostClassifier(verbose=0),
    "Gradient Boosting": GradientBoostingClassifier(),
    "AdaBoost": AdaBoostClassifier(),
    "K-Nearest Neighbors": KNeighborsClassifier(),
    "Decision Tree": DecisionTreeClassifier()
}
```

```
: # Helper function to calculate metrics for each model
def calculate_metrics(y_true, y_pred):
    precision = precision_score(y_true, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_true, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_true, y_pred, average='weighted', zero_division=0)
    return precision, recall, f1
```

```
: # Evaluate classifiers and populate model_accuracies
model_metrics = {}

for name, model in classifiers.items():
    pipeline = make_pipeline(StandardScaler(), MultiOutputClassifier(model, n_jobs=-1))
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)
    precision, recall, f1 = calculate_metrics(y_test, y_pred)
    model_metrics[name] = {'Precision': precision, 'Recall': recall, 'F1-Score': f1}
    print(f"\n{name} Metrics:\nPrecision: {precision:.4f}, Recall: {recall:.4f}, F1-Score: {f1:.4f}")

model_accuracies = {}

for name, model in classifiers.items():
    pipeline = make_pipeline(StandardScaler(), MultiOutputClassifier(model, n_jobs=-1))
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    model_accuracies[name] = acc
    print(f"{name} Accuracy: {acc:.4f}")
```

Figure 9: Model Performance Metrics

provided using these metrics. Recall measures the proportion of true positive predictions out of all positive predictions, whereas precision concentrates on predicting the portion of all the actual positive instances. If you are dealing with a case of imbalanced dataset, F1 score is great because of the output that it provides, it balances precision and recall together.

Confusion Matrices: To perform analysis over the predictions of each individual model, these were used to determine which target label predictions were true positives, false positives, true negatives, and false negatives. This helped me understand what labels models struggled to predict.

3.7 Results and Outputs

The results of the evaluation were presented in both textual and graphical formats: Accuracies, precisions, recalls and F1 scores were calculated and printed for each model. These metrics made it possible to substantially quantitatively compare the algorithms, and thus to identify which ones do well and where they fall short. The visual outputs of the project were essential for interpreting results:

Performance Metrics Comparison: Accuracy and F1-scores of each model were bar plotted to easily see the best performing algorithms.

```python
# Bar plot for accuracy scores
accuracy_scores = [model_accuracies[name] for name in model_accuracies]
plt.bar(model_accuracies.keys(), accuracy_scores, color='skyblue')
plt.title('Accuracy Scores for Models')
plt.ylabel('Accuracy')
plt.xticks(rotation=45)
plt.show()
```

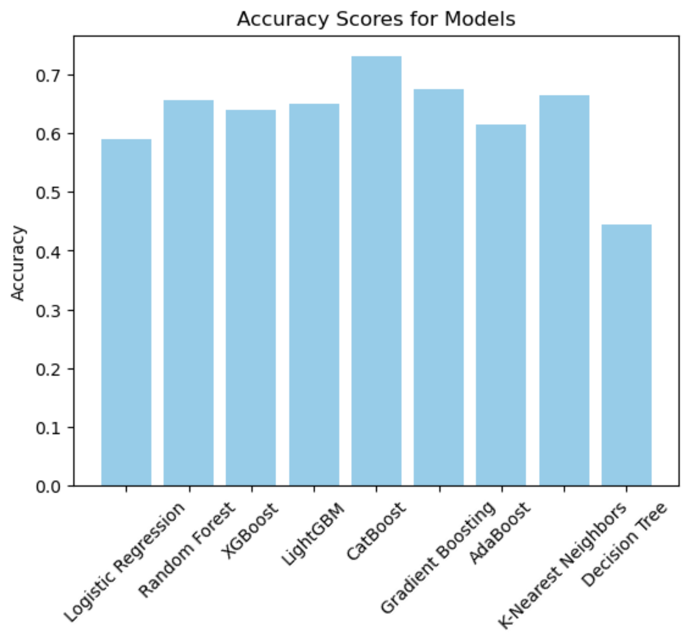


Figure 10: Accuracy Scores for Models

```python
# Metrics comparison plot
metrics_df = pd.DataFrame(model_metrics).T
metrics_df.plot(kind='bar', figsize=(10, 6))
plt.title('Precision, Recall, and F1-Score for Models')
plt.ylabel('Score')
plt.xticks(rotation=45)
plt.show()
```

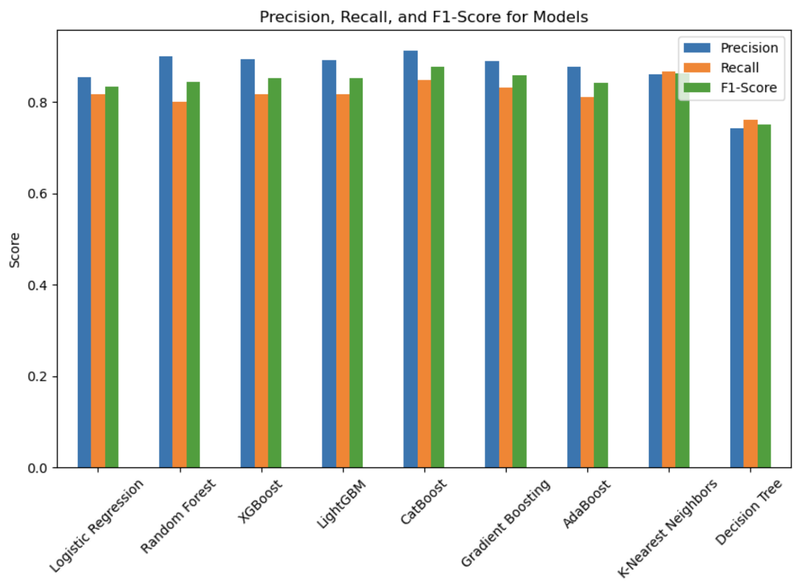


Figure 11: Precision, Recall and F1 Scores for Models

```python
# Pairwise comparison of Precision vs Recall
plt.scatter(metrics_df['Precision'], metrics_df['Recall'], c='red')
plt.title('Precision vs Recall for Models')
plt.xlabel('Precision')
plt.ylabel('Recall')
for i, txt in enumerate(metrics_df.index):
    plt.annotate(txt, (metrics_df['Precision'][i], metrics_df['Recall'][i]))
plt.show()
```

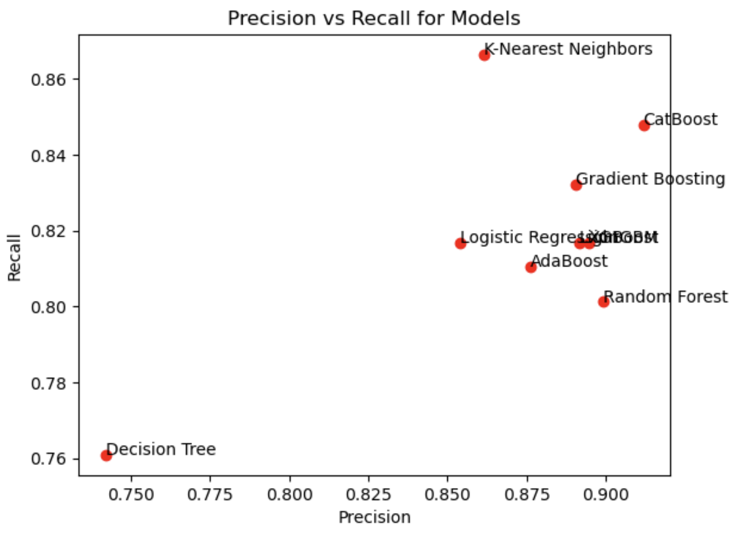
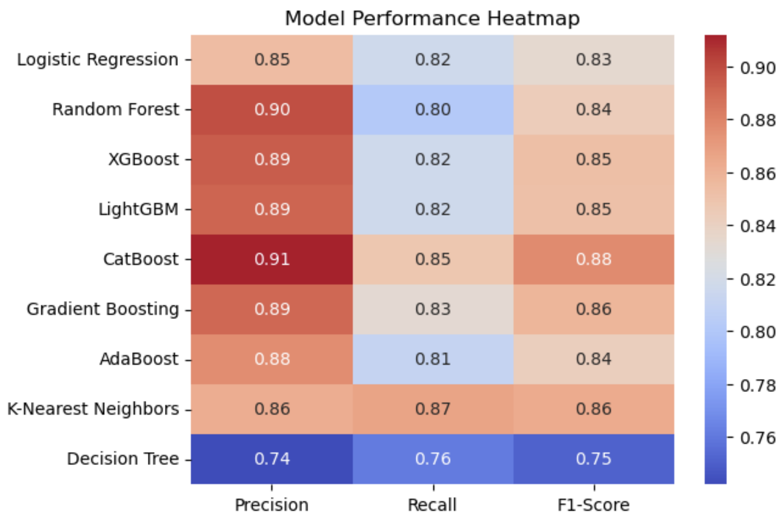


Figure 12: Precision vs Recall for Moddels

```python
[17]: # Heatmap for model performance
sns.heatmap(metrics_df, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Model Performance Heatmap')
plt.show()
```



	Precision	Recall	F1-Score
Logistic Regression	0.85	0.82	0.83
Random Forest	0.90	0.80	0.84
XGBoost	0.89	0.82	0.85
LightGBM	0.89	0.82	0.85
CatBoost	0.91	0.85	0.88
Gradient Boosting	0.89	0.83	0.86
AdaBoost	0.88	0.81	0.84
K-Nearest Neighbors	0.86	0.87	0.86
Decision Tree	0.74	0.76	0.75

Figure 13: Model Performance Heatmap

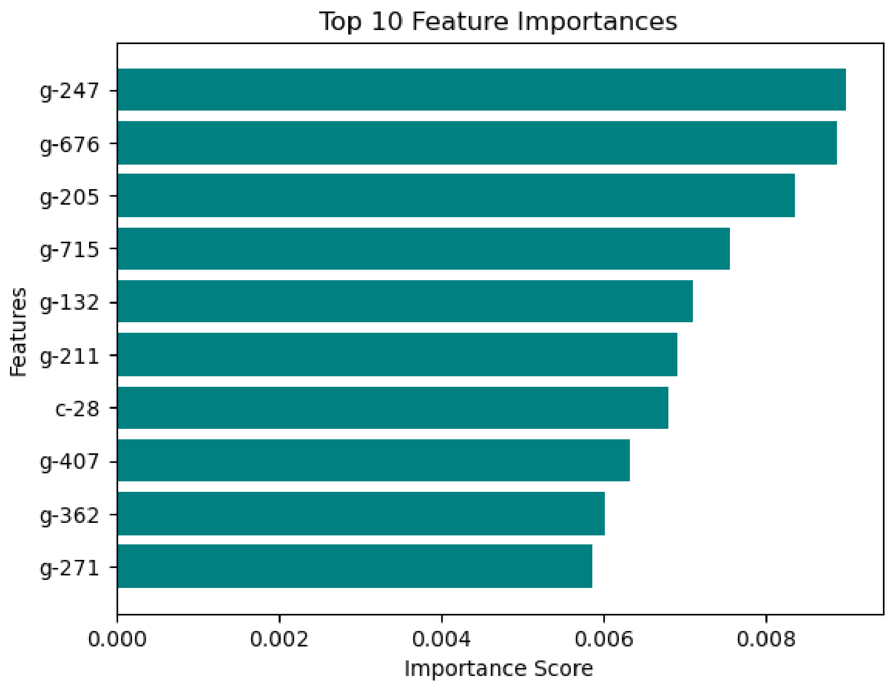


Figure 14: Feature Importance

Precision vs. Recall Scatter Plots: Specifically highlighting these was crucial in multi label classification work, as some labels may be under represented and thus some trade offs a model makes in tradeoff of precision and recall.
Heatmaps: These gave us the precision, recall, and F1-scores for all the models in order to understand their behavior better across different labels.

3.8 Feature Importance Analysis

Feature importance scores were calculated from Random Forest, XG Boost, Cat Boost and were visualized. For interpretability and perhaps input feature engineering improvements, bar plots of the top 10 most influential features provided insights into characteristics of the dataset.