ROUTE 25 - November

Ricardo Abreu

14.11.2025

Capgemini engineering

8.11 November 2025

Code available at: ricardo-abreu1/Route-25

8.11.1 Technical Activity – QoS Prediction Model Development

Change History

Date	Version	Author	Description
11/11/2025	1	Ricardo Abreu	Onboarding, 8.11.1 - Explorating QWS dataset
12/11/2025	2	Ricardo Abreu	8.11.1 - Data Exploratory Analysis, Data Processed. Implemented LinReg, Random Forest and XGBoost
13/11/2025	3	Ricardo Abreu	8.11.1 - Data Profiling, Exploratory Analysis, Data Processed (scripts improvements)
14/11/2025	4	Ricardo Abreu	8-11-1 - Models Tuning and Evaluation

Table of Contents

Contents

Week 1

Load and Inspect Data sources – Dataset “QWS Dataset V1”

```
# DataFrame creation using files avaiable
df_list = []

for file in files:
    try:
        df_list.append(pd.read_csv(file, sep=','))
    except Exception as e:
        print(f"Skipping {file}: {e}")

if df_list:
    df = pd.concat(df_list, ignore_index=True)
    print(df.shape)
else:
    print("No valid files")

df.head()
```

```
(364, 13)
```

	# Response Time	# Availability	# Throughput	# Successability	# Reliability
0	45.0	83	27.2	50	97.4
1	71.75	100	14.6	88	85.5
2	117.0	100	23.4	83	88.0
3	70.0	100	5.4	83	79.3
4	105.2	100	18.2	80	92.2

The data source was in **.txt format** and contained comments and non-standard headers, which required initial cleaning. In this first step, a copy of the file was created, and the first 42 lines—containing only the project description—were removed. Next, a header row with column names was added to match the data structure. The file was then loaded into the local environment for data analysis.

```
# datatype check
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 364 entries, 0 to 363
Data columns (total 13 columns):
 #   Column                                                      Non-Null Count  Dtype
---  ------                                                      --------------  -----
 0   Response Time                                               364 non-null    float64
 1   Availability                                                364 non-null    int64
 2   Throughput                                                  364 non-null    float64
 3   Successability                                              364 non-null    int64
 4   Reliability                                                 364 non-null    float64
 5   Compliance                                                  364 non-null    int64
 6   Best Practices                                              364 non-null    int64
 7   Latency                                                     364 non-null    float64
 8   Documentation                                               364 non-null    int64
 9   WsRF: Web Service Relevancy Function (%)                    364 non-null    int64
 10  Class: levels representing service offering qualities (1 through 4)  364 non-null    int64
 11  Service Name                                                364 non-null    object
 12  WSDL Address                                                364 non-null    object
dtypes: float64(4), int64(7), object(2)
memory usage: 37.1+ KB
```

The next phase consisted of **exploratory data analysis** to gain a deep understanding of the data, metadata, and dataset structure.

```
# Column statistics
df.describe(include='all')
```

	# Response Time	# Availability	# Throughput	# Successability
count	364.0	364.0	364.0	364.0
unique	*Missing value*	*Missing value*	*Missing value*	*Missing value*
top	*Missing value*	*Missing value*	*Missing value*	*Missing value*
freq	*Missing value*	*Missing value*	*Missing value*	*Missing value*
mean	840.2830302197801	84.75274725274726	7.284890109890109	64.43681318681318
std	2764.317553466436	20.45149790446885	6.458114014711211	21.17287798736948
min	45.0	14.0	0.1	7.0
25%	136.785	74.75	2.1750000000000003	50.0
50%	236.65	96.0	5.6	67.0
75%	480.0625	100.0	10.625	80.25

11 rows x 13 cols 10 per page Page 1 of 2

3. Missing values and duplicated values analysis

```
# Missing values
missing = df.isnull().sum()
missing_pct = (missing / len(df)) * 100

df_nulls = pd.DataFrame({
    'Column': df.columns,
    'Missing Values': missing.values,
    'Missing %': missing_pct.values
})

# Print duplicates separately
duplicates = df.duplicated().sum()
print('*' * 24)
print(f"Total duplicated rows: {duplicates}")
print('*' * 24)

# Sort by missing percentage
df_nulls = df_nulls.sort_values(by='Missing %', ascending=False)
df_nulls
```

```
************************
Total duplicated rows: 0
************************
```

	Column	# Missing Values	# Missing %
0	Response Time	0	0.0
1	Availability	0	0.0
2	Throughput	0	0.0

After that, the data was inspected for **null values, duplicates, and outliers**, following standard procedures for dataset profiling.

A) DataFrame without outliers

```
# Using IQR method
Q1 = df_numerical.quantile(0.25)
Q3 = df_numerical.quantile(0.75)
IQR = Q3 - Q1

# Filter outliers
df_no_outliers = df[~((df_numerical < (Q1 - 1.5 * IQR)) | (df_numerical > (Q3 + 1.5 * IQR))).any(axis=1)]

print(df_no_outliers.shape)
df_no_outliers
```

As a final note for this stage, the following points were observed:

1. Data types are correct;
2. The dataset contains no duplicates or null values;
3. The dataset includes some outliers; however, given its small size, these were retained.

4. The target variable chosen for prediction is **“WsRF: Web Service Relevancy Function (%)”**.

Univariate Analysis (1 variable)

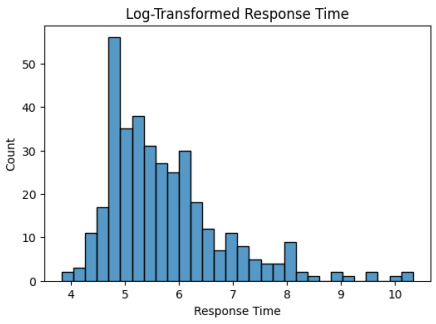
```
# Check for skewness or log-transform suggestion
print("Skewness of numeric columns:")
print(df_numerical.skew())
```

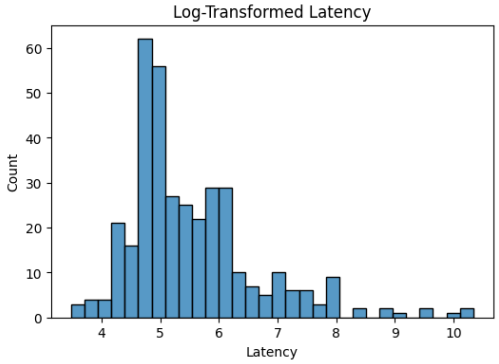
```
Skewness of numeric columns:
Response Time                                          7.928711
Availability                                          -1.478005
Throughput                                             1.241758
Successability                                        -0.741783
Reliability                                           -0.645650
Compliance                                             0.634805
Best Practices                                        -0.966659
Latency                                                8.063425
Documentation                                          0.113207
WsRF: Web Service Relevancy Function (%)              -0.307966
Class: levels representing service offering qualities (1 through 4)   -0.273701
dtype: float64
```

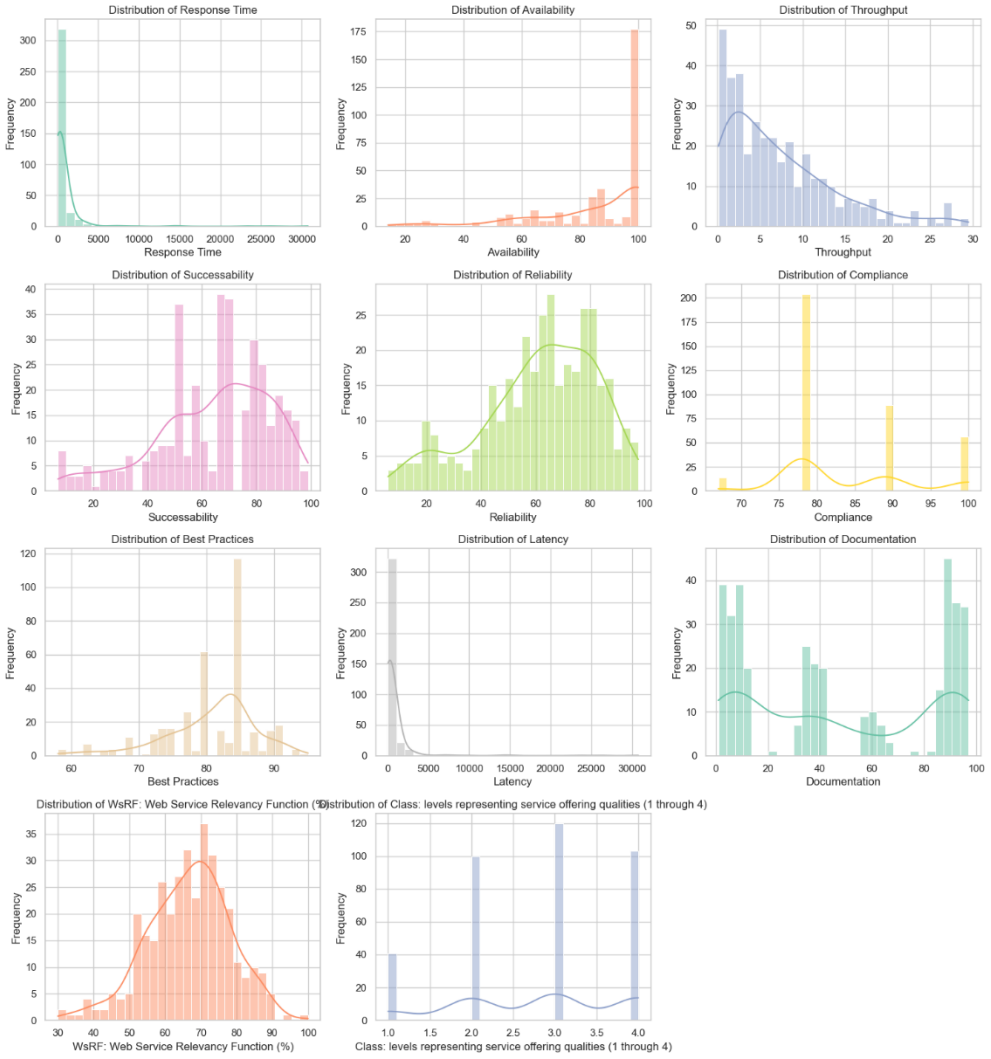
- Response Time and Latency are skewed (distribution of data is not symmetric)

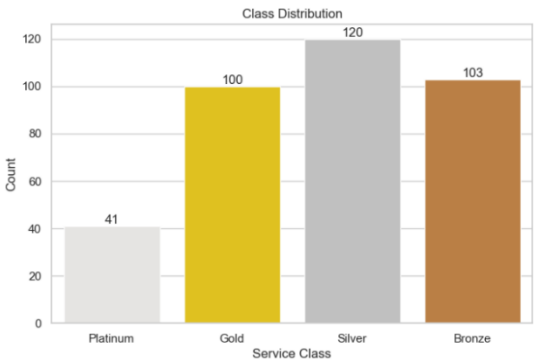
In this phase, a **graphical analysis** was performed to better understand the data distribution and relationships between variables. Univariate, bivariate, and multivariate analyses were conducted, considering service classification into predefined categories.

Initially, the variables **“Response Time”** and **“Latency”** were explored due to their less symmetric distributions. Subsequently, the remaining numeric variables were analyzed according to their distribution.





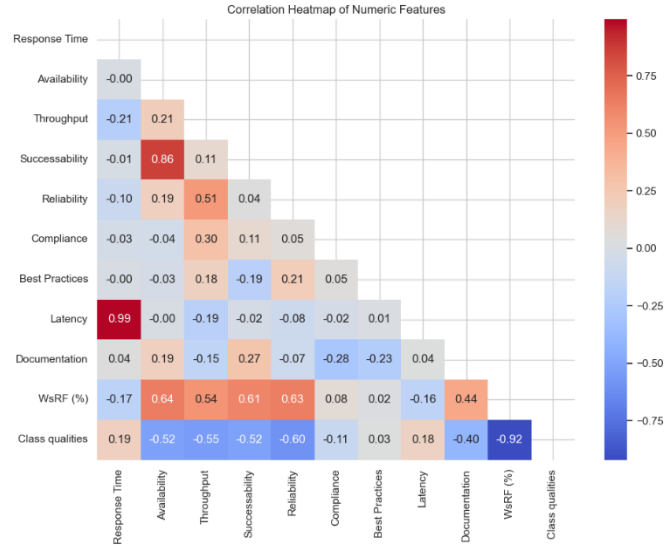




The next step involved creating a **correlation matrix** during the bivariate analysis to identify patterns among variables.

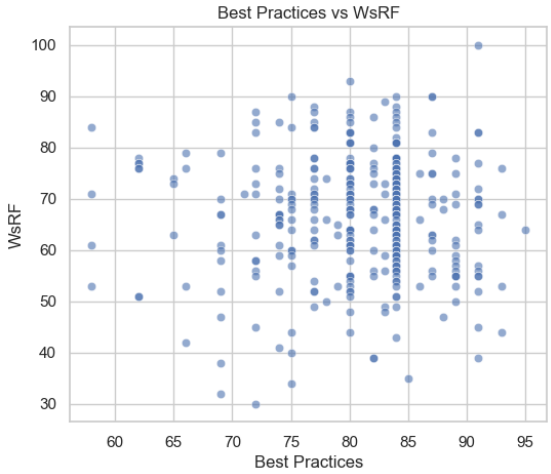
Bivariate Analysis (2 variables)

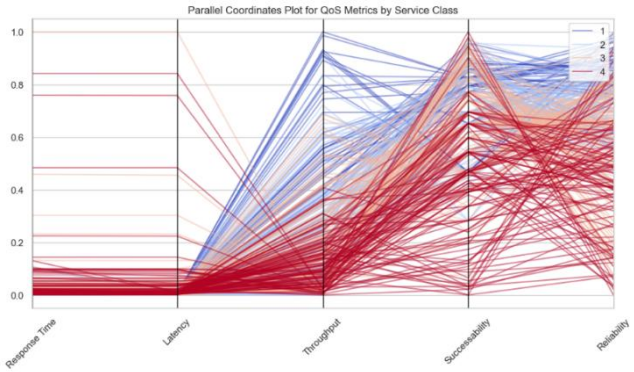
```
df_num = df_numerical.copy()
df_num.rename(columns={"Class: levels representing service offering qualities (1 through 4)": 'Class qualities', 'WsRF: Web Service Relevancy Function (%)': 'WsRF (%)'}, inplace=True)

# Compute correlation matrix
corr_matrix = df_num.corr()

# Plot heatmap
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, mask=mask, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap of Numeric Features')
plt.tight_layout()
plt.show()
```



Below is an example of a chart illustrating the decision to drop a variable from the model because it showed no meaningful pattern with the target variable.

```
# Response Time vs Latency
plt.figure(figsize=(6, 5))
sns.scatterplot(x='Response Time', y='Latency', data=df_num, alpha=0.6)
plt.title('Response Time vs Latency')
plt.xlabel('Response Time')
plt.ylabel('Latency')
plt.show()

# Throughput vs WsRF
plt.figure(figsize=(6, 5))
sns.scatterplot(x='Best Practices', y='WsRF (%)', data=df_num, alpha=0.6)
plt.title('Best Practices vs WsRF')
plt.xlabel('Best Practices')
plt.ylabel('WsRF')
plt.show()

# Throughput vs WsRF
plt.figure(figsize=(6, 5))
sns.scatterplot(x='Throughput', y='WsRF (%)', data=df_num, alpha=0.6)
plt.title('Throughput vs WsRF')
plt.xlabel('Throughput')
plt.ylabel('WsRF')
plt.show()
```





Finally, a brief multivariate analysis was performed to understand how certain characteristics relate to service level.

Key findings from this analysis:

- Univariate Analysis:
  - Availability is concentrated near 100%, suggesting most services are highly available.
  - Successability, Reliability, and WsRF are fairly normal, centered around 50–80.
- Bivariate Analysis:
  - Platinum services (Class 1) generally have lower Response Time and Latency compared to Bronze.
  - Throughput tends to decrease as class level increases (Platinum → Bronze).
  - Reliability and Successability are higher for Platinum and Gold.
  - Response Time and Latency show strong positive correlation (services with high response time also have high latency).
  - Best Practices vs WsRF shows weak correlation, suggesting WsRF may not strongly depend on Best Practices.
  - Response Time and Latency have the highest correlation among numeric features.

- Availability and Documentation have low correlation with other metrics, indicating limited predictive power.

- Multivariate Analysis:
  - Platinum services cluster with low Response Time and Latency, high Reliability and Successability.
  - Bronze services show the opposite pattern: high delays and lower reliability.
  - Columns that can be dropped - (Latency or Response Time, only one) and Best Practices.

Machine Learning model making:

In this phase, **Machine Learning (ML) models** were built to train on the data and predict future datasets.
The workflow followed this sequence:

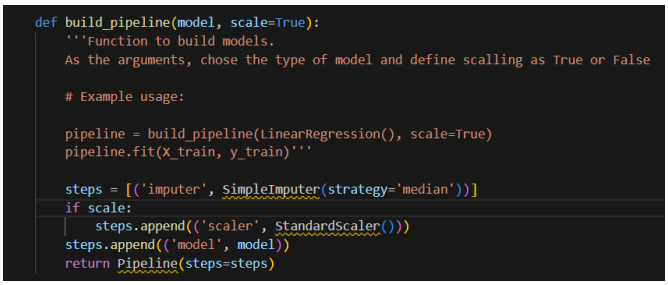
1. Split the data into training and test sets.
2. Define preprocessing steps. For the Linear Regression model, the data suffers an additional step of “scaling“ so that the features had a similar distribution.
3. Train the model using the training set.
4. Evaluate the model on the test set.
5. Tune the model (an iterative process to identify the best parameters for improved performance).

Data Splitting:

The figure below illustrates how data was split and columns with low correlation were removed (to reduce noise in the model).

```
# 1. Drop leakage columns
leakage_cols = ['Service Name', 'WSDL Address', 'Latency', 'Best Practices']
df = df.drop(columns=[col for col in leakage_cols if col in df.columns])

# 2. Validate features
qos_features = ['Response Time', 'Availability', 'Throughput', 'Successability', 'Reliability', 'Compliance', 'Documentation']
missing_features = [col for col in qos_features if col not in df.columns]
if missing_features:
    raise KeyError(f"Missing features: {missing_features}")

X = df[qos_features].copy()

# 3. Validate target
target_col = 'WsRF: Web Service Relevancy Function (%)'
if target_col not in df.columns:
    raise KeyError(f"Target column '{target_col}' not found.")
y = df[target_col].copy()

# 4. First split: Train (70%) and Remaining (30%)
X_train, X_remaining, y_train, y_remaining = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# 5. Split remaining: Test (70%) and Validation (30%)
X_test, X_val, y_test, y_val = train_test_split(
    X_remaining, y_remaining, test_size=0.3, random_state=42
)

# 6. Save training data
train_data = pd.concat([X_train, y_train], axis=1)
train_data.to_csv('qws1_dataset/training_data_regression.csv', index=False)

# 7. Save test data
test_data = pd.concat([X_test, y_test], axis=1)
test_data.to_csv('qws1_dataset/test_data_regression.csv', index=False)

# 8. Save validation data
val_data = pd.concat([X_val, y_val], axis=1)
val_data.to_csv('qws1_dataset/validation_data_regression.csv', index=False)

print(f"Train: {X_train.shape}, Test: {X_test.shape}, Validation: {X_val.shape}")
```

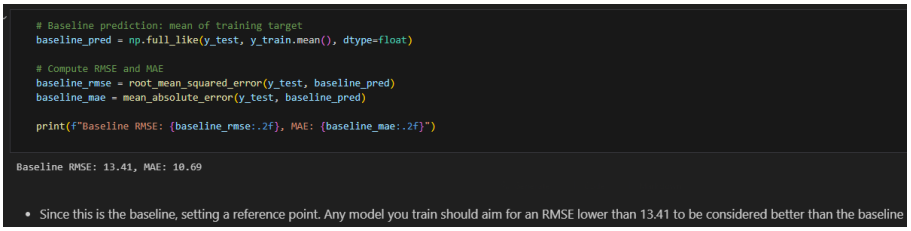
```
Train: (254, 7), Test: (77, 7), Validation: (33, 7)
```



```python
def build_pipeline(model, scale=True):
    '''Function to build models.
    As the arguments, chose the type of model and define scalling as True or False

    # Example usage:

    pipeline = build_pipeline(LinearRegression(), scale=True)
    pipeline.fit(X_train, y_train)'''

    steps = [('imputer', SimpleImputer(strategy='median'))]
    if scale:
        steps.append(('scaler', StandardScaler()))
    steps.append(('model', model))
    return Pipeline(steps=steps)
```

The data was split into 3 distinct subsets. The first subset is the training data that will be used to train the ML model, the second is test data to test the ML model according the certain metrics and finally the third subset is the validation data that it wont be used in this task but in a different task in the future for back testing and data drifts.

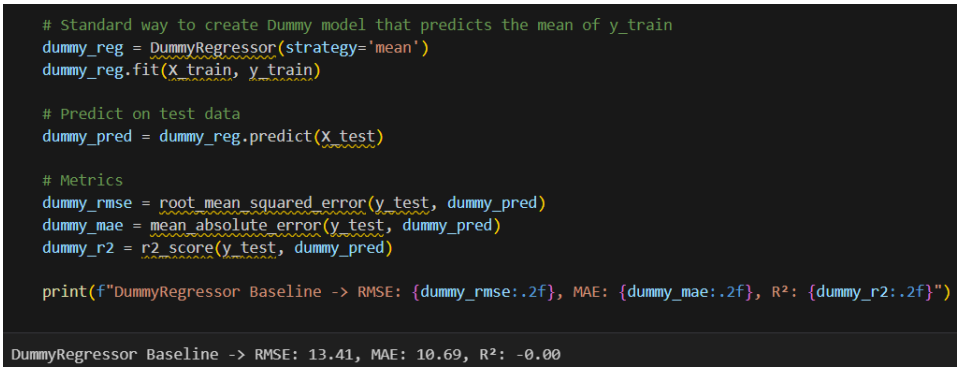
Next, a **baseline model** was defined for comparison with subsequent models. There is 2 ways to do this. The first picture shows the manual way to compute metrics for the baseline model and the second figure shows how to do it according to scikit-learn.



```python
# Baseline prediction: mean of training target
baseline_pred = np.full_like(y_test, y_train.mean(), dtype=float)

# Compute RMSE and MAE
baseline_rmse = root_mean_squared_error(y_test, baseline_pred)
baseline_mae = mean_absolute_error(y_test, baseline_pred)

print(f"Baseline RMSE: {baseline_rmse:.2f}, MAE: {baseline_mae:.2f}")
```

```
Baseline RMSE: 13.41, MAE: 10.69
```

- Since this is the baseline, setting a reference point. Any model you train should aim for an RMSE lower than 13.41 to be considered better than the baseline



```python
# Standard way to create Dummy model that predicts the mean of y_train
dummy_reg = DummyRegressor(strategy='mean')
dummy_reg.fit(X_train, y_train)

# Predict on test data
dummy_pred = dummy_reg.predict(X_test)

# Metrics
dummy_rmse = root_mean_squared_error(y_test, dummy_pred)
dummy_mae = mean_absolute_error(y_test, dummy_pred)
dummy_r2 = r2_score(y_test, dummy_pred)

print(f"DummyRegressor Baseline -> RMSE: {dummy_rmse:.2f}, MAE: {dummy_mae:.2f}, R²: {dummy_r2:.2f}")
```

```
DummyRegressor Baseline -> RMSE: 13.41, MAE: 10.69, R²: -0.00
```

The models selected were **Linear Regression, Random Forest, and XGBoost**.

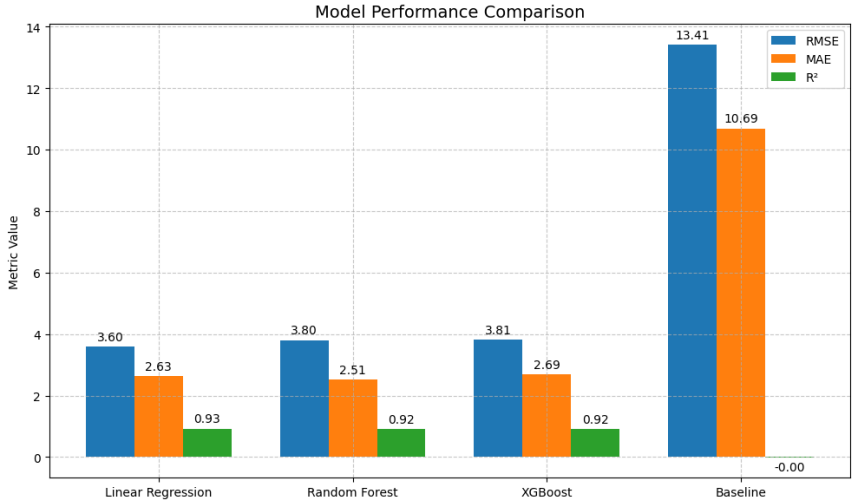
```python
# Define models
models = {
    'LinearRegression': LinearRegression(),
    'RandomForest': RandomForestRegressor(random_state=42),
    'XGBoost': XGBRegressor(random_state=42)
}

results = {}

for name, model in models.items():
    # Scale only for LinearRegression
    scale = True if name == 'LinearRegression' else False
    pipeline = build_pipeline(model, scale=scale)

    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)

    rmse = np.sqrt(root_mean_squared_error(y_test, y_pred))
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    results[name] = {'RMSE': rmse, 'MAE': mae, 'R²': r2}

# Display results
for model_name, metrics in results.items():
    print(f"{model_name}: RMSE={metrics['RMSE']:.2f}, MAE={metrics['MAE']:.2f}, R²={metrics['R²']:.3f}")

results_df = pd.DataFrame(results).T.reset_index()
results_df.columns = ['Model', 'RMSE', 'MAE', 'R²']
results_df
```

```
Model Performance:
```

	Model	# RMSE	# MAE	# R²
0	Linear Regression	3.5986977509976765	2.6328531950906426	0.927806724857026
1	Random Forest	3.8037618366018973	2.5113698630136994	0.9193447515586426
2	XGBoost	3.8115673065185547	2.6854708194732666	0.9190133810043335
3	Baseline	13.408000765060446	10.687379372028431	-0.002153048229032839

The following table shows the results of 3 metrics for all 3 models:

Model	RMSE	MAE	R^2
Linear Regression	3.5986977509976765	2.6328531950906426	0.927806724857026
Random Forest	3.8037618366018973	2.5113698630136994	0.9193447515586426
XGBoost	3.8115673065185547	2.6854708194732666	0.9190133810043335

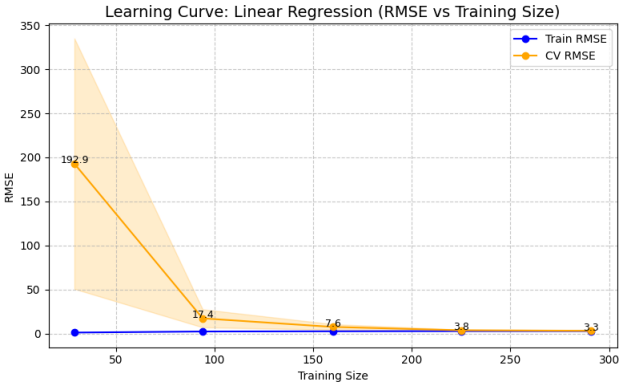


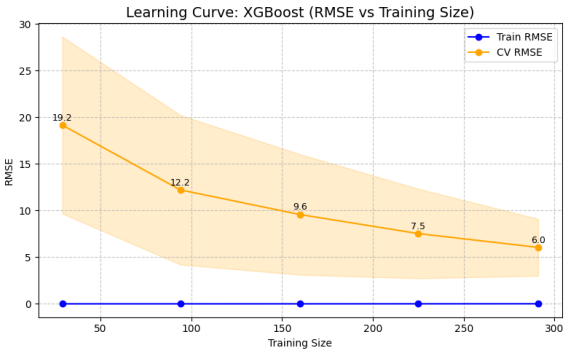
Confidence Interval

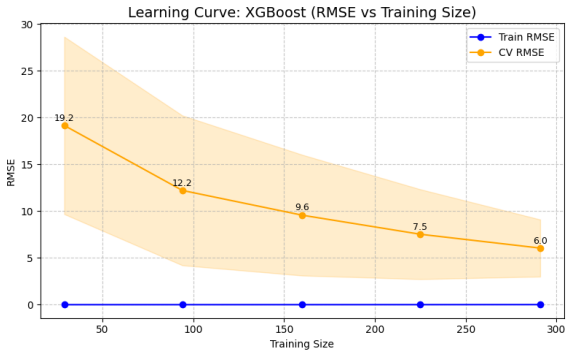
A confidence interval gives a range of values that likely contains the true metric for the model, based on testing data. For it was applied bootstrapping. Bootstrapping simulates many possible test sets by resampling testing data.

2.3.1 Bootstrapping to calculate a confidence interval (CI):

```python
def bootstrap_ci(metric_func, y_true, y_pred, n_bootstrap=1000):
    metrics = []
    for _ in range(n_bootstrap):
        idx = np.random.choice(len(y_true), len(y_true), replace=True)
        metrics.append(metric_func(y_true[idx], y_pred[idx]))
    return np.percentile(metrics, [2.5, 97.5])

# Compute CI for each model
for name, y_pred in preds.items():
    rmse_ci = bootstrap_ci(root_mean_squared_error, y_test.values, y_pred)
    mae_ci = bootstrap_ci(mean_absolute_error, y_test.values, y_pred)
    r2_ci = bootstrap_ci(r2_score, y_test.values, y_pred)
    print(f"{name} -> RMSE 95% CI: {rmse_ci}, MAE 95% CI: {mae_ci}, R² 95% CI: {r2_ci}")
```







1. Linear Regression (benefits greatly from more data, overfitting is evident at small sample sizes)

- Train RMSE (blue): Starts near zero and stays extremely low across all training sizes.

- The model fits training data almost perfectly, which is expected for linear regression on small datasets.

- CV RMSE (orange): Starts very high (~193) with small training size, then drops sharply to ~3.3 at full size.

- Gap between train and CV RMSE at small sizes > severe overfitting initially.

- Trend: As training size increases, CV RMSE decreases and stabilizes but never meets train RMSE.

2. Random Forest (highly flexible and overfits small datasets, dding data improves CV RMSE but gap persists)

- Train RMSE (blue): Very low (~1.5) and flat across all sizes > model memorizes training data.

- CV RMSE (orange): Starts at ~20.6, decreases gradually to ~6.0 at full size.

- Shaded region: Wide at small sizes, narrows as size grows > variance reduces with more data.

3. XGBoost (Overfits small datasets, Still needs regularization or more data to close the gap)

- Train RMSE (blue): Near zero across all sizes > extreme memorization.

- CV RMSE (orange): Starts at ~19.2, drops to ~6.0, overfitting similar to Random Forest but better CV RMSE at mid-range sizes.

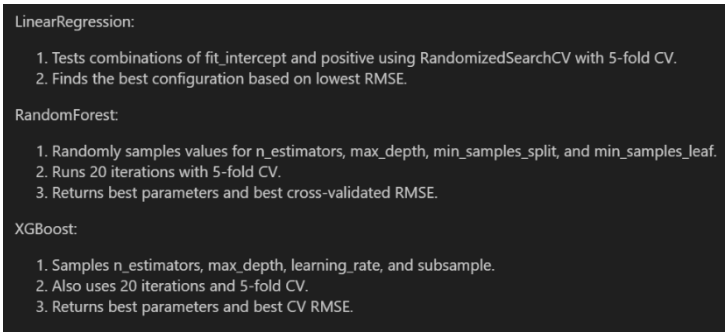
- Variance: High at small sizes, narrows with more data.

Model Tuning - RandomizedSearchCV

Nesta secção é descrito como os modelos foram melhorados. Cada modelo tem um algoritmo com hyper-parâmetros (parâmetros do algoritmo) que podem ser modificados no sentido de melhorar a performance por parte do modelo de ML. Segue o exemplo abaixo para descrever parâmetros que podem ser mudados na Regressão Linear:

Eg. Hyperparameters for Linear Regression:

- - fit_intercept: Whether to calculate the intercept (True/False).
- - positive: Force coefficients to be positive (True/False).
- - normalize: Deprecated in newer versions (use StandardScaler instead).
- - n_jobs: For parallel computation (usually not critical here)



3.1 Linear Regression

```
# Parameter grid
param_grid_lr = {
    'model__fit_intercept': [True, False],
    'model__positive': [True, False]
}

# Pipeline with scaling
lr_pipeline = build_pipeline(LinearRegression(), scale=True)

# RandomizedSearchCV
lr_search = RandomizedSearchCV(
    estimator=lr_pipeline,
    param_distributions=param_grid_lr,
    n_iter=4,
    cv=5,
    scoring='neg_root_mean_squared_error',
    random_state=42
)

lr_search.fit(X_train, y_train)

print("Best Params:", lr_search.best_params_)
print("Best CV RMSE:", -lr_search.best_score_)
```

3.2 Random Forest

```
# Parameter grid
param_grid_rf = {
    'model__n_estimators': randint(50, 300),
    'model__max_depth': randint(3, 20),
    'model__min_samples_split': randint(2, 10),
    'model__min_samples_leaf': randint(1, 5)
}

# Pipeline (no scaling)
rf_pipeline = build_pipeline(RandomForestRegressor(random_state=42), scale=False)

# RandomizedSearchCV
rf_search = RandomizedSearchCV(
    estimator=rf_pipeline,
    param_distributions=param_grid_rf,
    n_iter=20,
    cv=5,
    scoring='neg_root_mean_squared_error',
    random_state=42,
    n_jobs=-1
)

rf_search.fit(X_train, y_train)

print("Best Params:", rf_search.best_params_)
print("Best CV RMSE:", -rf_search.best_score_)
```

```
Best Params: {'model__max_depth': 14, 'model__min_samples_leaf': 1, 'model__min_samples_split': 2, 'model__n_estimators': 268}
Best CV RMSE: 3.089152416950278
```

3.3 XGBoost

```
# Parameter grid
param_grid_xgb = {
    'model__n_estimators': randint(100, 500),
    'model__max_depth': randint(3, 10),
    'model__learning_rate': [0.01, 0.05, 0.1, 0.2],
    'model__subsample': [0.6, 0.8, 1.0]
}

# Pipeline (no scaling)
xgb_pipeline = build_pipeline(XGBRegressor(random_state=42), scale=False)

# RandomizedSearchCV
xgb_search = RandomizedSearchCV(
    estimator=xgb_pipeline,
    param_distributions=param_grid_xgb,
    n_iter=20,
    cv=5,
    scoring='neg_root_mean_squared_error',
    random_state=42,
    n_jobs=-1
)

xgb_search.fit(X_train, y_train)

print("Best Params:", xgb_search.best_params_)
print("Best CV RMSE:", -xgb_search.best_score_)
```

```
Best Params: {'model__learning_rate': 0.05, 'model__max_depth': 4, 'model__n_estimators': 291, 'model__subsample': 0.6}
Best CV RMSE: 2.3284844160079956
```

Afterwards, the train data was fed into the models to predict values.

4. Prediction

```
best_lr = lr_search.best_estimator_
best_rf = rf_search.best_estimator_
best_xgb = xgb_search.best_estimator_

# Predict and evaluate
for name, model in [('Linear Regression', best_lr), ('Random Forest', best_rf), ('XGBoost', best_xgb)]:
    y_pred = model.predict(X_test)
    rmse = root_mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print(f"{name}: RMSE={rmse:.2f}, MAE={mae:.2f}, R²={r2:.3f}")
```

```
Linear Regression: RMSE=3.60, MAE=2.63, R²=0.928
Random Forest: RMSE=3.83, MAE=2.51, R²=0.918
XGBoost: RMSE=2.75, MAE=1.80, R²=0.958
```

1. Extract best models
2. Evaluate tuned models on test set

After calling the function to predict the values, all of the aspects related to evaluating the models were made.

Comparison

```
# ---- Baseline ----
baseline_pred = np.full_like(y_test, y_train.mean(), dtype=float)
baseline_rmse = root_mean_squared_error(y_test, baseline_pred)
baseline_mae = mean_absolute_error(y_test, baseline_pred)
baseline_r2 = r2_score(y_test, baseline_pred)

previous_results = results_df.copy()
previous_results['Model'] = previous_results['Model'] + ' (Untuned)'

# ---- Tuned Results ----
tuned_data = []
for name, model in [('LinearRegression', best_lr), ('RandomForest', best_rf), ('XGBoost', best_xgb)]:
    y_pred = model.predict(X_test)
    rmse = root_mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    tuned_data.append({'Model': f'{name} (Tuned)', 'RMSE': rmse, 'MAE': mae, 'R²': r2})

tuned_results = pd.DataFrame(tuned_data)

# ---- Baseline Row ----
baseline_row = pd.DataFrame([{'Model': 'Baseline', 'RMSE': baseline_rmse, 'MAE': baseline_mae, 'R²': baseline_r2}])

# ---- Combine All ----
comparison_df = pd.concat([baseline_row, previous_results, tuned_results], ignore_index=True)

print("\nComparison Table:")
comparison_df
```

Comparison Table:

	Model	# RMSE	# MAE	# R²
0	Baseline	13.408000765060446	10.687379372028431	-0.002153048229032839
1	Linear Regression (Untuned)	3.5986977509976765	2.6328531950906426	0.927806724857026
2	Random Forest (Untuned)	3.8037618366018973	2.5113698630136994	0.9193447515586426
3	XGBoost (Untuned)	3.8115673065185547	2.6854708194732666	0.9190133810043335
4	Linear Regression (Tuned)	3.5986977509976765	2.6328531950906426	0.927806724857026
5	Random Forest (Tuned)	3.8263169789963936	2.513851973011654	0.918385393789493
6	XGBoost (Tuned)	2.7543282508850098	1.8014215230941772	0.9577100276947021

Then it was produced a sample of the dataframe with actual values and the ones predicted by each model.

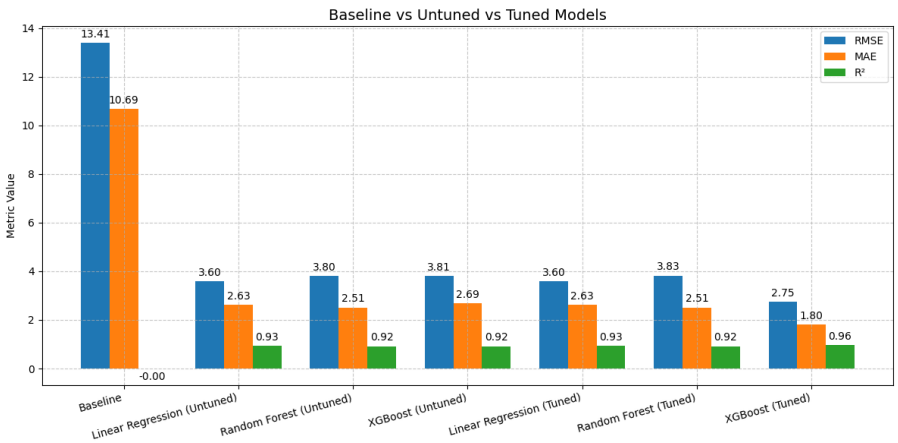
```python
# Predictions DataFrame
preds_df_tuned = pd.DataFrame({'Actual Values': y_test})
for model_name, y_pred in tuned_preds.items():
    preds_df_tuned[model_name] = y_pred
preds_df_tuned['Baseline'] = baseline_pred

print("\nActual vs Predictions:")
preds_df_tuned.head()
```

```
Actual vs Predictions:
```

	# Actual Values	# Linear Regression	# Random Forest	# XGBoost	# Baseline
193	67	65.92621949256372	67.38432835820896	67.42043	66.52920962199313
33	82	82.0939694857598	80.53358208955224	81.377106	66.52920962199313
15	86	86.4372476843109	86.06716417910448	86.48468	66.52920962199313
347	45	51.098416727870884	50.29477611940298	48.74173	66.52920962199313
57	77	75.56233919828183	74.48134328358209	75.072	66.52920962199313

Afterwards, all metrics of all models were graphically put together for visibility,

```python
# Remove "Baseline (Untuned)" from the DataFrame
plot_df = comparison_df[comparison_df['Model'] != 'Baseline (Untuned)']

# Plot
metrics = ['RMSE', 'MAE', 'R²']
x = np.arange(len(plot_df['Model']))
width = 0.25

fig, ax = plt.subplots(figsize=(12, 6))
bars = []
for i, metric in enumerate(metrics):
    bar_container = ax.bar(x + i*width, plot_df[metric], width, label=metric)
    bars.append(bar_container)

# Add labels
for bar_container in bars:
    ax.bar_label(bar_container, fmt='%.2f', padding=3)

ax.set_xticks(x + width)
ax.set_xticklabels(plot_df['Model'], rotation=15, ha='right')
ax.grid(True, linestyle='--', alpha=0.7)
ax.set_ylabel('Metric Value')
ax.set_title('Baseline vs Untuned vs Tuned Models', fontsize=14)
ax.legend()
plt.tight_layout()
plt.show()
```



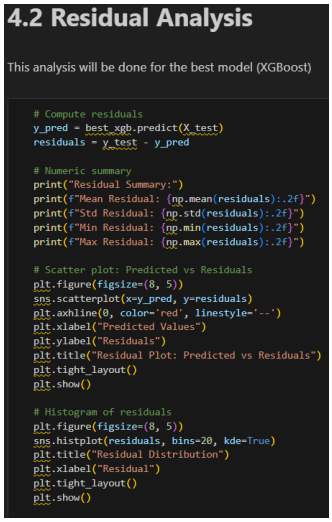
- **RMSE (Root Mean Squared Error):** Measures prediction error magnitude. Lower is better.

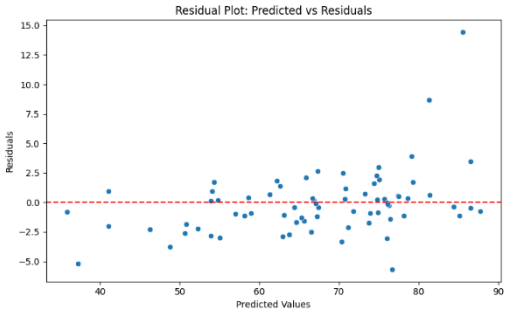
- **MAE (Mean Absolute Error):** Measures average absolute deviation. Useful for interpretability.
- **R^2 (Coefficient of Determination):** Indicates how much variance in the target is explained by the model. Closer to 1 means better fit.

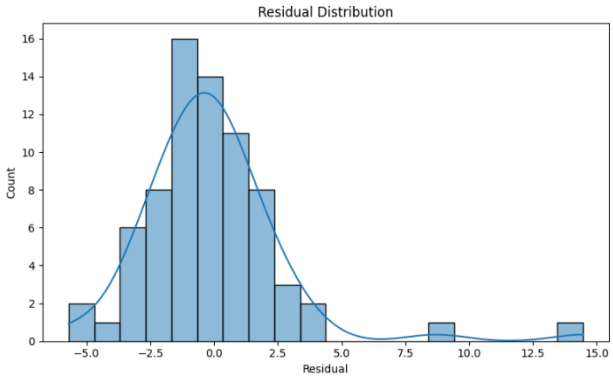
Then, the results from the tuning of the models, the results from the models that were not tuned and the baseline model were compared, as showed in the previous picture. It's results are in the following table:

Model	RMSE	MAE	R^2
Baseline	13.408000765060446	10.687379372028431	-0.002153048229032839
Linear Regression	3.5986977509976765	2.6328531950906426	0.927806724857026
Random Forest	3.8037618366018973	2.5113698630136994	0.9193447515586426
XGBoost	3.8115673065185547	2.6854708194732666	0.9190133810043335
Linear Regression (tuned)	3.5986977509976765	2.6328531950906426	0.927806724857026
Random Forest (tuned)	3.8263169789963936	2.513851973011654	0.918385393789493
XGBoost (tuned)	2.7543282508850098	1.8014215230941772	0.9577100276947021

This next code block produced the following:







For the Residual Plot (Predicted vs Residuals):

- Centering around zero: Most residuals cluster near the red line (0), which is good — predictions are generally unbiased.
- Spread increases slightly for higher predicted values: At predicted values above ~75, residuals become more positive and more variable.
- This suggests mild heteroscedasticity (variance of errors grows with predicted value).
- Outliers: There are a few large positive residuals (up to ~15), meaning the model underpredicted those cases significantly.

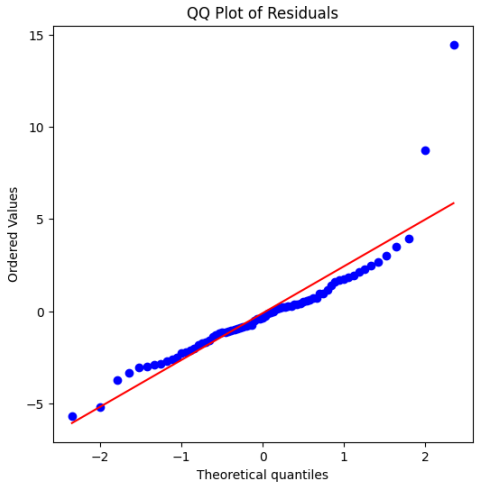
Residual Distribution (Histogram):

- Shape: Mostly bell-shaped and centered near zero, which is ideal for regression assumptions.
- Slight right skew: A tail toward positive residuals indicates some cases where the model underestimates the target.
- No severe multimodality: No clear clusters, so the model likely captures the main pattern well.

QQ plot:

The following picture shows the code block for the QQ Plot.

```python
# QQ Plot
plt.figure(figsize=(6, 6))
stats.probplot(residuals, dist="norm", plot=plt)
plt.title("QQ Plot of Residuals")
plt.show()
```



Central Alignment:

- Most points lie close to the red line in the center, meaning residuals are approximately normal for the bulk of the data.

Heavy Right Tail (Positive Outliers):

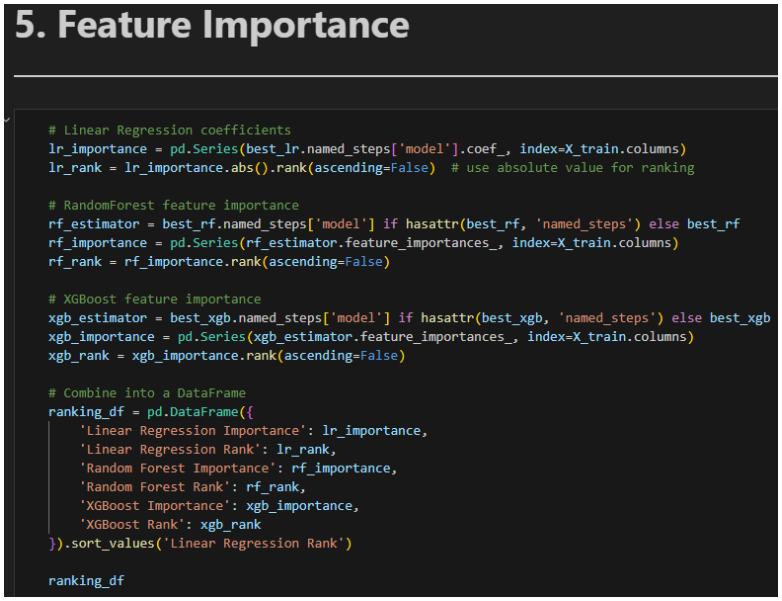
- The top-right points deviate strongly from the line, especially two extreme points around 9 and 15.
- This indicates large positive residuals. Model underpredicted these observations significantly (normal because since the amount of data is small the outliers were kept)

Slight Left Tail Deviation:

- Bottom-left points also deviate, but less severely than the right tail.
- Suggests some negative outliers (overpredictions), but not as extreme.

Feature Importance (Regression models):

In this section it's described the steps to find the importance of the features for each model. So, the next 2 figures explain how this was done, and then presented a table with the features and their respective importance for the models ordered by the Linear Regression Ranking.



	# Linear Regression Impor...	# Linear Regression Rank	# Random Forest Importa...	# Random Forest Rank	# XGBoost Importance	# XGBoost Rank
Reliability	5.452228963573292	1.0	0.18158429134569956	3.0	0.15469317	3.0
Documentatic	4.752199388763976	2.0	0.12365363118483816	6.0	0.109820016	6.0
Successability	3.5311796395723176	3.0	0.2062553969579572	1.0	0.21101622	2.0
Throughput	2.9655735953404303	4.0	0.1455769912336929	5.0	0.12479077	4.0
Availability	1.5570660834917547	5.0	0.15050121207790704	4.0	0.26688626	1.0
Response Tim	-0.9367468060359003	6.0	0.18589991935251302	2.0	0.11513554	5.0
Compliance	0.5433203401316606	7.0	0.006528557847392177	7.0	0.017658034	7.0

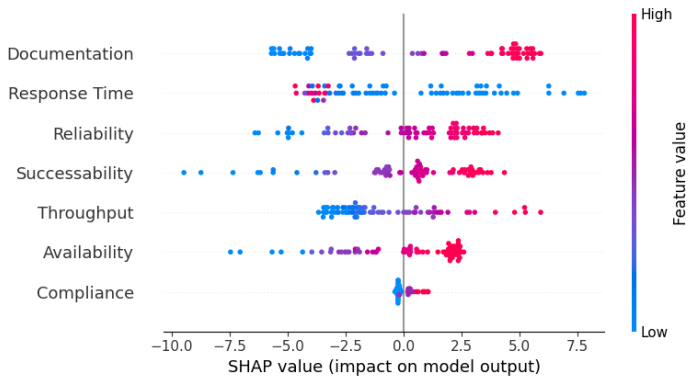
Week 2

To finish the script of the regression models, a final graph was constructed showing the features importances according to a Shap graph for the model with the best results which was XGBoost.

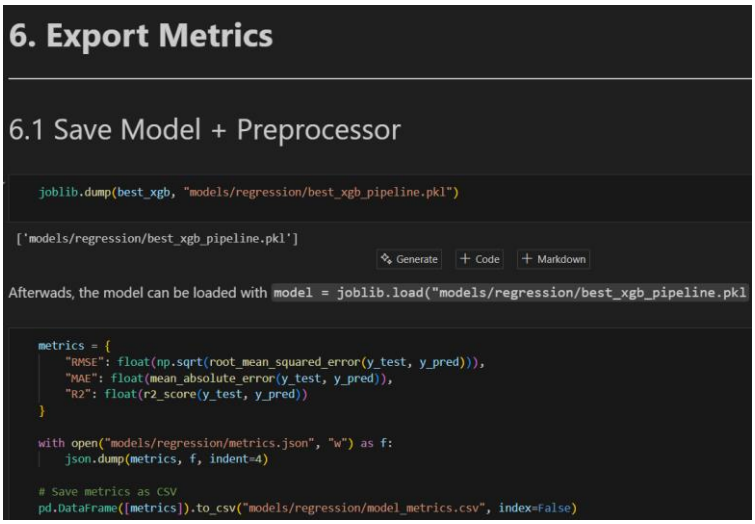
5.1 SHAP Interpretation

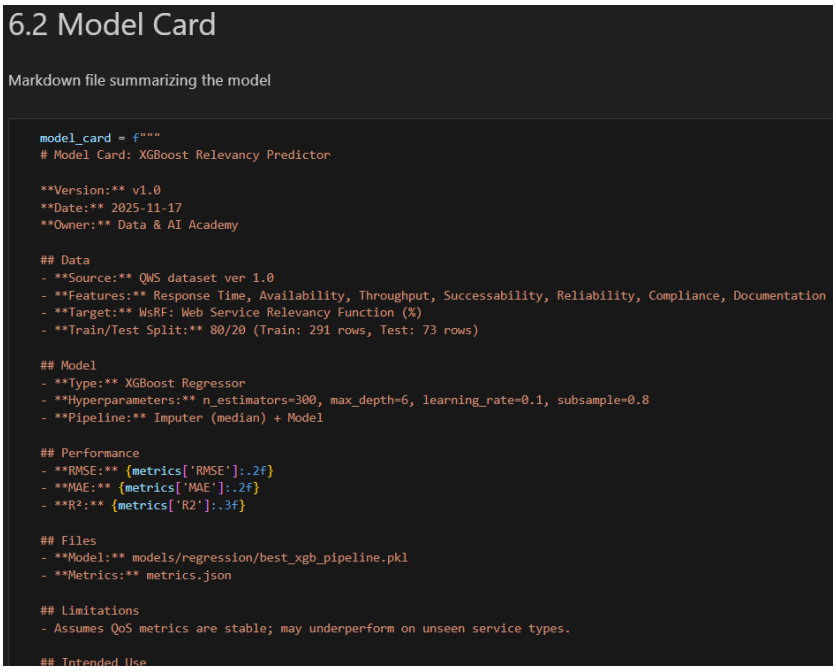
- Positive SHAP value → pushes prediction up.
- Negative SHAP value → pushes prediction down.

```
# Use the model inside pipeline
model = best_xgb.named_steps['model']

explainer = shap.Explainer(model, X_train)
shap_values = explainer(X_test)

# Summary plot
shap.summary_plot(shap_values, X_test)
```



Finally, the models were exported, with all the metrics for evaluation for json and csv format, and also a md file was created depicting at a high level the model and its metrics as showed in the following figures.





Classification Models

To construct a script for a categorical model, the process is almost the same as the regression, so the script follows almost the same structure. One of the key differences in this script is that the label (target value) which currently is a class between 1 through 4 already in the datatype integer). In this example it is not necessary but for consistency it suffered a process of encodement just in case any data was different than integer. Then this column fits these values and assigns values from 0 through 3, this is important because some ML Models work with 0 as a starting point.

```
# Non-feature columns and the highly correlated columns dropped
drop_cols = ['Service Name', 'WSDL Address', 'Class: levels representing service offering qualities (1 through 4)',
             'WsRF: Web Service Relevancy Function (%)']
feature_cols = [col for col in df.columns if col not in drop_cols]

X = df[feature_cols]
y = df['Class: levels representing service offering qualities (1 through 4)']

# Encoded target
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

print("Classes:", label_encoder.classes_)
```

Following a similar matter, the data was split into 3 subsets (training data, test data and validation data). This procedure is similar to the regression models depicted earlier.

```
# 3. First split: Train (70%) and Remaining (30%)
X_train, X_remaining, y_train, y_remaining = train_test_split(
    X, y_encoded,
    test_size=0.3,
    stratify=y_encoded,
    random_state=42
)

# 4. Split remaining: Test (70%) and Validation (30%)
X_test, X_val, y_test, y_val = train_test_split(
    X_remaining, y_remaining,
    test_size=0.3,
    stratify=y_remaining,
    random_state=42
)

# 5. Save training data
train_data = pd.concat([X_train.reset_index(drop=True),
                        pd.Series(y_train, name='Class').reset_index(drop=True)],
                       axis=1)

train_data.to_csv('qws1_dataset/training_data_classification.csv', index=False)

# 6. Save test data
test_data = pd.concat([X_test.reset_index(drop=True),
                       pd.Series(y_test, name='Class').reset_index(drop=True)],
                      axis=1)

test_data.to_csv('qws1_dataset/test_data_classification.csv', index=False)

# 7. Save validation data
val_data = pd.concat([X_val.reset_index(drop=True),
                      pd.Series(y_val, name='Class').reset_index(drop=True)],
                     axis=1)

val_data.to_csv('qws1_dataset/validation_data_classification.csv', index=False)

# 8. Shapes and class distribution
print(f"Train shape: {X_train.shape}, Test shape: {X_test.shape}, Validation shape: {X_val.shape}")
print("Class distribution in Train:", {i: sum(y_train == i) for i in set(y_train)})
print("Class distribution in Test:", {i: sum(y_test == i) for i in set(y_test)})
print("Class distribution in Validation:", {i: sum(y_val == i) for i in set(y_val)})
```

```
Train shape: (254, 9), Test shape: (77, 9), Validation shape: (33, 9)
Class distribution in Train: {np.int64(0): np.int64(28), np.int64(1): np.int64(70), np.int64(2): np.int64(84), np.int64(3): np.int64(72)}
Class distribution in Test: {np.int64(0): np.int64(9), np.int64(1): np.int64(21), np.int64(2): np.int64(25), np.int64(3): np.int64(22)}
Class distribution in Validation: {np.int64(0): np.int64(4), np.int64(1): np.int64(9), np.int64(2): np.int64(11), np.int64(3): np.int64(9)}
```

Then a baseline model was established to predict the most frequent class.

2.1 Baseline

- This is the baseline evaluation, setting a reference point for future models. Any model trained should aim for an Accuracy, F1 and ROC-AUC higher to be better than the baseline.

```
# Dummy classifier
baseline_clf = DummyClassifier(strategy='most_frequent')
baseline_clf.fit(X_train, y_train)
baseline_pred = baseline_clf.predict(X_test)

# Accuracy
baseline_acc = accuracy_score(y_test, baseline_pred)

# F1 Score
baseline_f1 = f1_score(y_test, baseline_pred, average='macro')

# ROC-AUC
baseline_prob = baseline_clf.predict_proba(X_test)
y_test_bin = label_binarize(y_test, classes=[0, 1, 2, 3])
baseline_roc_auc = roc_auc_score(y_test_bin, baseline_prob, average='macro', multi_class='ovr')

print(f"Baseline (Majority Class) Accuracy: {baseline_acc:.4f}")
print(f"Baseline F1 Score (Macro): {baseline_f1:.4f}")
print(f"Baseline ROC-AUC (Macro): {baseline_roc_auc:.4f}")
```

Just to have an idea of the predicted values of the classification models chosen, the following pipeline was constructed, and the values are in the following figures.

```python
# Models
models = {
    'Logistic Regression': Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', LogisticRegression(max_iter=1000, random_state=42))
    ]),
    'Random Forest': Pipeline([
        ('classifier', RandomForestClassifier(random_state=42))
    ]),
    'XGBoost': Pipeline([
        ('classifier', XGBClassifier(eval_metric='mlogloss', random_state=42))
    ])
}

results = {}

y_test_bin = label_binarize(y_test, classes=[0, 1, 2, 3])
# Baseline metrics
results['Baseline'] = {
    'Accuracy': baseline_acc,
    'Precision': baseline_precision,
    'Recall': baseline_recall,
    'F1': baseline_f1,
    'ROC-AUC': baseline_roc_auc
}

predictions_df = pd.DataFrame({'Actual Values': y_test})

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)

    # Predictions for each model
    predictions_df[name] = y_pred

    # Metrics
    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average='macro')
    roc_auc = roc_auc_score(y_test_bin, y_prob, average='macro', multi_class='ovr')
    precision = precision_score(y_test, y_pred, average='macro')
```

```python
    recall = recall_score(y_test, y_pred, average='macro')

    results[name] = {
        'Accuracy': acc,
        'Precision': precision,
        'Recall': recall,
        'F1': f1,
        'ROC-AUC': roc_auc
    }

# Baseline predictions
predictions_df['Baseline'] = baseline_pred

print("\nPredictions vs Actual:")
predictions_df
```

```
Predictions vs Actual:
```

	# Actual Values	# Logistic Regression	# Random Forest	# XGBoost	# Baseline
0	0	0	0	0	2
1	3	3	3	3	2
2	2	2	2	2	2
3	3	3	3	3	2
4	0	0	0	0	2
5	3	3	3	3	2
6	3	2	2	2	2
7	2	2	2	2	2
8	3	3	3	3	2
9	1	1	1	0	2

Following the same procedure for the regression models, these were evaluated according to 5 of the following metrics.

2.2.2 Evaluation

It will be used 5 metrics:

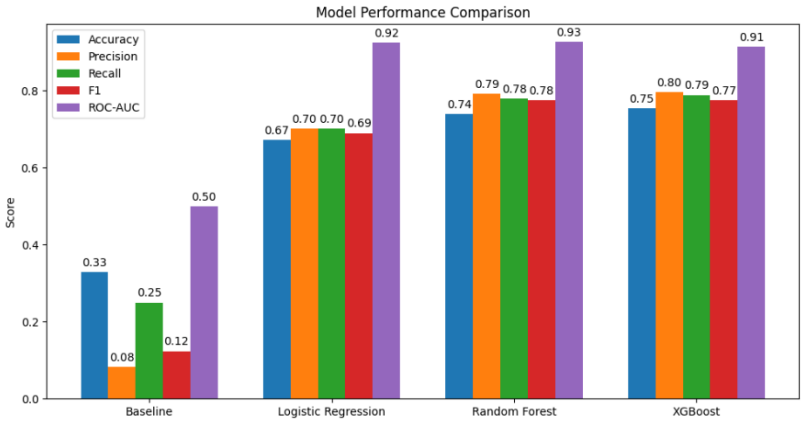
- Accuracy: proportion of predictions that the model got right
- Recall: proportion of positive cases that the model identified correctly
- Precision: proportion of predicted positive cases where the true label is actually positive
- F1: overall metric that combines recall and precision
- Receiver Operating Characteristic – Area Under the Curve (ROC-AUC): shows how well a model can separate positive and negative classes

```python
# DataFrame with metrics
results_df = pd.DataFrame(results).T.reset_index()
results_df.columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1', 'ROC-AUC']

results_df
```

```
✓ 0.0s  Open 'results_df' in Data Wrangler
```

	Model	# Accuracy	# Precision	# Recall	# F1	# ROC-AUC
0	Baseline	0.3287671232876712	0.0821917808219178	0.25	0.1237113402061855 7	0.5
1	Logistic Regression	0.6712328767123288	0.7020050125313283	0.7011904761904761	0.6899470899470899	0.9241977306161943
2	Random Forest	0.7397260273972602	0.7918650793650794	0.7797619047619048	0.7751750700280112	0.9261779185549402
3	XGBoost	0.7534246575342466	0.7953525641025641	0.7886904761904763	0.7746326600815765	0.9133036269660258

```python
metrics = ['Accuracy', 'Precision', 'Recall', 'F1', 'ROC-AUC']
models_names = results_df['Model']
values = [results_df[m] for m in metrics]

x = np.arange(len(results_df))
width = 0.15

fig, ax = plt.subplots(figsize=(12, 6))
for i, metric in enumerate(metrics):
    bars = ax.bar(x + i*width, values[i], width, label=metric)

    for bar in bars:
        height = bar.get_height()
        ax.annotate(f'{height:.2f}',
                    xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0, 3),
                    textcoords="offset points",
                    ha='center', va='bottom')

ax.set_xticks(x + width*2)
ax.set_xticklabels(models_names)
ax.set_ylabel('Score')
ax.set_title('Model Performance Comparison')
ax.legend()
plt.show()
```



These next sections are for the tuning of hyperparameters for all models.

3. Hyperparameter tuning

3.1 Random Forest

```python
# Parameter grid
param_dist = {
    'n_estimators': [100, 200, 300, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

rf = RandomForestClassifier(random_state=42, class_weight='balanced')

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rf,
    param_distributions=param_dist,
    n_iter=50,
    scoring={'f1_macro':'f1_macro','accuracy':'accuracy','roc_auc_ovr':'roc_auc_ovr'},
    refit='f1_macro',
    cv=cv_strategy,
    verbose=2,
    random_state=42,
    n_jobs=-1,
    return_train_score=True
)

random_search.fit(X_train, y_train)

print("Best Params:", random_search.best_params_)
print("Best Score (CV F1):", random_search.best_score_)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Params: {'n_estimators': 500, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_depth': 20, 'bootstrap': True}
Best Score (CV F1): 0.8053853068211902
```

3.2 Logistic Regression

```
# Parameter grid
param_grid_lr = {
    'classifier__penalty': ['l2'],
    'classifier__C': [0.01, 0.1, 1, 10, 100],
    'classifier__solver': ['lbfgs', 'saga']
}

lr_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression(max_iter=1000, random_state=42, class_weight='balanced'))
])

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# RandomizedSearchCV
lr_search = RandomizedSearchCV(
    estimator=lr_pipeline,
    param_distributions=param_grid_lr,
    n_iter=20,
    scoring={'f1_macro':'f1_macro','accuracy':'accuracy','roc_auc_ovr':'roc_auc_ovr'},
    refit='f1_macro',
    cv=cv_strategy,
    random_state=42,
    n_jobs=-1,
    return_train_score=True
)

lr_search.fit(X_train, y_train)

print("Best Params (LogisticRegression):", lr_search.best_params_)
print("Best CV F1:", lr_search.best_score_)
```

```
c:\Users\riabreu\OneDrive - Capgemini\Desktop\Capgemini\Route-25\venv\Lib\site-packages\sklearn\model_selection\_sear
  warnings.warn(
Best Params (LogisticRegression): {'classifier__solver': 'lbfgs', 'classifier__penalty': 'l2', 'classifier__C': 100}
Best CV F1: 0.8824433152703277
```

3.3 XGBoost

```
param_grid_xgb = {
    'n_estimators': [100, 200, 300, 500],
    'max_depth': [3, 5, 7, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0],
    'gamma': [0, 0.1, 0.3, 0.5]
}

xgb = XGBClassifier(eval_metric='mlogloss', random_state=42)

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

xgb_search = RandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_grid_xgb,
    n_iter=50,
    scoring={'f1_macro':'f1_macro','accuracy':'accuracy','roc_auc_ovr':'roc_auc_ovr'},
    refit='f1_macro',
    cv=cv_strategy,
    verbose=2,
    random_state=42,
    n_jobs=-1,
    return_train_score=True
)

xgb_search.fit(X_train, y_train)

print("Best Params (XGBoost):", xgb_search.best_params_)
print("Best CV F1:", xgb_search.best_score_)
```

After tunning the models, by settings hyperparameters that influence the model, a cross-validation was made to identify the best model overall based on F1 metric.

3.4 Cross-Validation for Best Model

```
all_models = {
    'Baseline': baseline_clf,
    'Logistic Regression Untuned': models['Logistic Regression'],
    'Random Forest Untuned': models['Random Forest'],
    'XGBoost Untuned': models['XGBoost'],
    'Logistic Regression Tuned': lr_search.best_estimator_,
    'Random Forest Tuned': random_search.best_estimator_,
    'XGBoost Tuned': xgb_search.best_estimator_
}

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

cv_results = {}
for name, model in all_models.items():
    scores = cross_validate(
        model,
        X,
        y_encoded,
        cv=cv_strategy,
        scoring=['accuracy', 'precision_macro', 'recall_macro', 'f1_macro', 'roc_auc_ovr']
    )
    cv_results[name] = {
        'Accuracy': scores['test_accuracy'].mean(),
        'Precision': scores['test_precision_macro'].mean(),
        'Recall': scores['test_recall_macro'].mean(),
        'F1': scores['test_f1_macro'].mean(),
        'ROC-AUC': scores['test_roc_auc_ovr'].mean(),
        'Accuracy_std': scores['test_accuracy'].std(),
        'F1_std': scores['test_f1_macro'].std()
    }

cv_df = pd.DataFrame(cv_results).T.reset_index()
cv_df.columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1', 'ROC-AUC', 'Accuracy_std', 'F1_std']
cv_df = cv_df.sort_values(by='F1', ascending=False)

print("\nCross-Validation Ranking (mean ± std):")
for _, row in cv_df.iterrows():
    print(f"{row['Model']}: F1={row['F1']:.4f} ± {row['F1_std']:.4f}, Accuracy={row['Accuracy']:.4f} ± {row['Accuracy_std']:.4f}")

best_model_name = cv_df.iloc[0]['Model']
best_model = all_models[best_model_name]
print(f"\nBest model after cross-validation: {best_model_name}")
```

```
Cross-Validation Ranking (mean ± std):
Logistic Regression Tuned: F1=0.8469 ± 0.0506, Accuracy=0.8405 ± 0.0543
Logistic Regression Untuned: F1=0.8201 ± 0.0501, Accuracy=0.8158 ± 0.0499
Random Forest Untuned: F1=0.7983 ± 0.0341, Accuracy=0.8021 ± 0.0341
Random Forest Tuned: F1=0.7883 ± 0.0318, Accuracy=0.7966 ± 0.0287
XGBoost Tuned: F1=0.7761 ± 0.0559, Accuracy=0.7884 ± 0.0400
XGBoost Untuned: F1=0.7597 ± 0.0422, Accuracy=0.7747 ± 0.0271
Baseline: F1=0.1240 ± 0.0005, Accuracy=0.3297 ± 0.0018

Best model after cross-validation: Logistic Regression Tuned
```

The best-performing model based on F1 score is Logistic Regression Tuned, and the cross-validation results indicate strong and consistent performance across all metrics:

- Accuracy (0.8405 ± 0.0543) -> The model correctly predicts about 84% of cases on average. The ±0.05 shows moderate variability across folds, which is acceptable for this dataset size.
- Precision_macro (0.8438) -> When the model predicts a class, it is correct 84% of the time on average across all classes, indicating reliable predictions.
- Recall_macro (0.8549) -> The model captures about 85% of actual cases, showing strong sensitivity and ability to identify true positives.
- F1_macro (0.8469 ± 0.0506) -> A balanced measure of precision and recall. At ~85%, the model demonstrates excellent overall performance.
- ROC_AUC_OVR (0.9669) -> Outstanding ability to separate classes (close to 1.0). This suggests the model is highly effective at distinguishing between different service quality levels.

For results, it was shown the best model was the tuned Logistic Regression the following details:

- Accuracy (0.8405 ± 0.0543) -> The model correctly predicts about 84% of cases on average. The ±0.05 shows moderate variability across folds, which is acceptable for this dataset size.
- Precision_macro (0.8438) -> When the model predicts a class, it is correct 84% of the time on average across all classes, indicating reliable predictions.
- Recall_macro (0.8549) -> The model captures about 85% of actual cases, showing strong sensitivity and ability to identify true positives.
- F1_macro (0.8469 ± 0.0506) -> A balanced measure of precision and recall. At ~85%, the model demonstrates excellent overall performance.
- ROC_AUC_OVR (0.9669) -> Outstanding ability to separate classes (close to 1.0). This suggests the model is highly effective at distinguishing between different service quality levels.

In these 2 next figures, it's showed an example of predicted values by every model and the actual values and the metrics for all models respectively.

It is important to note that, in each day the code was reviewed and optimized some sections according not only to the current task but also for future ones. Nontheless in this report it's showed everything for transparency and to show the progress of the author.

3.4.1 Predicted vs Actual Values

```python
# Predictions DataFrame
predictions_df = pd.DataFrame({'Actual': y_test})

# Baseline predictions
predictions_df['Baseline'] = baseline_pred

# Untuned models predictions
for name, model in models.items():
    predictions_df[f'{name}_Untuned'] = model.predict(X_test)

# Tuned models predictions
tuned_models = {
    'LogisticRegression_Tuned': lr_search.best_estimator_,
    'RandomForest_Tuned': random_search.best_estimator_,
    'XGBoost_Tuned': xgb_search.best_estimator_
}

for name, model in tuned_models.items():
    predictions_df[name] = model.predict(X_test)

predictions_df.head()
```

0.1s Open 'predictions_df' in Data Wrangler

	# Actual	# Baseline	# Logistic Regression_Unt...	# Random Forest_Untuned	# XGBoost_Untuned	# LogisticRegression_Tuned	# RandomForest_Tuned	# XGBoost_Tuned
0	0	2	0	0	0	0	0	0
1	3	2	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2
3	3	2	3	3	3	3	3	3
4	0	2	0	0	0	0	0	0

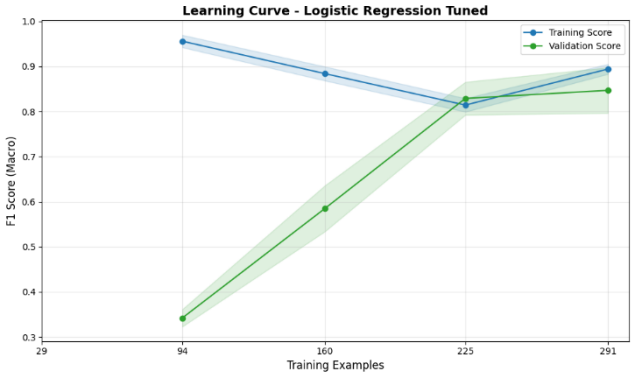
```python
cv_df = pd.DataFrame(cv_results).T.reset_index()
cv_df.columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1', 'ROC-AUC', 'Accuracy_std', 'F1_std']
cv_df = cv_df.sort_values(by='Accuracy', ascending=False)

# Remover colunas std
cv_df = cv_df.drop(columns=['Accuracy_std', 'F1_std'])
cv_df
```

0.0s Open 'cv_df' in Data Wrangler

	Model	# Accuracy	# Precision	# Recall	# F1	# ROC-AUC
4	Logistic Regression Tuned	0.8405251141552512	0.8437873315697637	0.8548809523809522	0.8469062720888193	0.9668936750475771
1	Logistic Regression Untuned	0.8158295281582953	0.8464933917791975	0.8102380952380951	0.8201330735558171	0.9632554941198178
2	Random Forest Untuned	0.8020547945205478	0.8305321417821971	0.793452380952381	0.798256499382743	0.9508972600931095
5	Random Forest Tuned	0.796613394216134	0.8217003995042763	0.7820833333333334	0.7882816939468807	0.9477925511584804
6	XGBoost Tuned	0.7883561643835616	0.8211077175634236	0.7700992063492065	0.7761258108241798	0.9401789625048131
3	XGBoost Untuned	0.7746955859969559	0.807411912570001	0.7520039682539682	0.7596726672111755	0.9436204476385681
0	Baseline	0.3296803652968036	0.08242009132420091	0.25	0.1239690721649484	0.5

As the regression model, it was made a graph with a learning curve for the tuned logistic regression classification model.

```python
# Learning Curve for Best Model
cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

train_sizes, train_scores, test_scores = learning_curve(
    best_model, X, y_encoded, cv=cv_strategy, scoring='f1_macro', n_jobs=-1
)

# Medians and standard deviation
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_mean, 'o-', color='#1f77b4', label='Training Score')
plt.plot(train_sizes, test_mean, 'o-', color='#2ca02c', label='Validation Score')

# Confidence area
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.15, color='#1f77b4')
plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, alpha=0.15, color='#2ca02c')

# Ajustments
plt.title(f'Learning Curve - {best_model_name}', fontsize=14, fontweight='bold')
plt.xlabel('Training Examples', fontsize=12)
plt.ylabel('F1 Score (Macro)', fontsize=12)
plt.xticks(train_sizes)
plt.grid(alpha=0.3)
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



The learning curve illustrates how the F1 score (macro) evolves as the training set size increases:

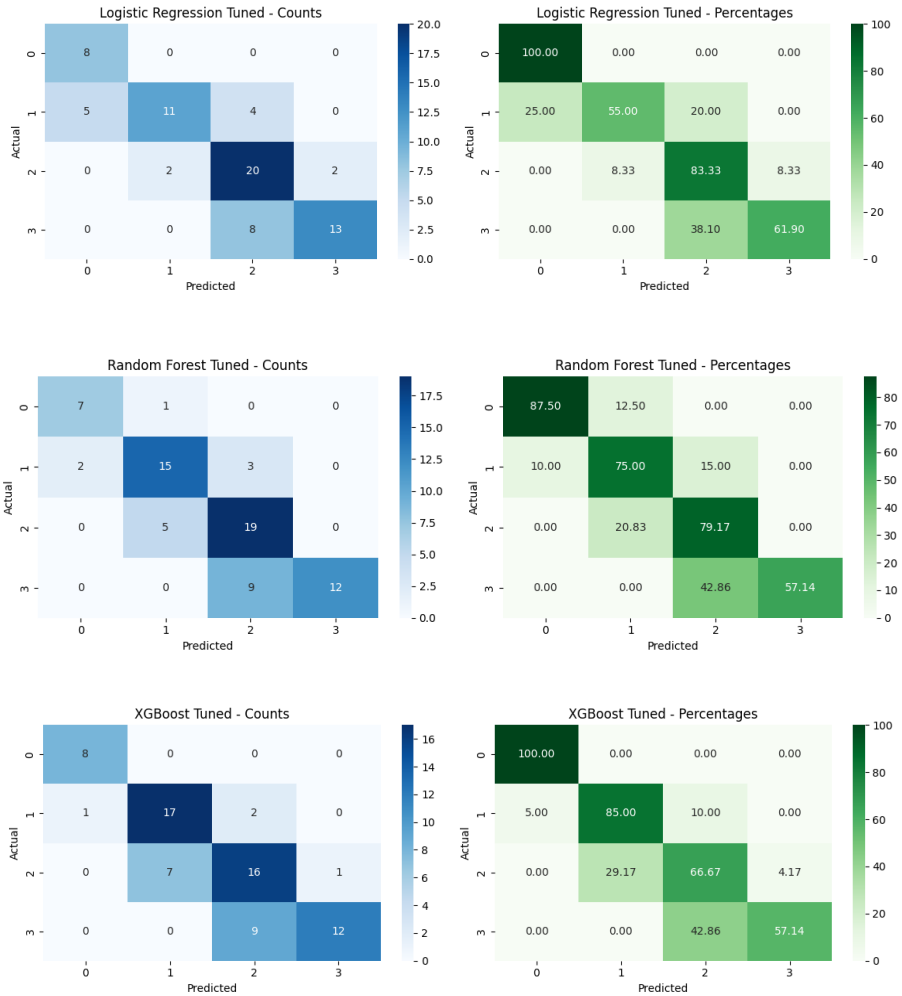
- Initial Phase (≈94 samples): The training score starts very high (≈0.95), while the validation score is low (≈0.34). This large gap indicates strong overfitting when the model is trained on a small dataset — it memorizes the training data but fails to generalize.
- Intermediate Phase (≈160–225 samples): As more data is added, the validation score improves significantly (rising to ≈0.82), while the training score decreases slightly (≈0.81). This convergence suggests the model is learning more generalizable patterns and reducing overfitting.
- Final Phase (≈291 samples): Both curves stabilize close together (training ≈0.90, validation ≈0.85). The small remaining gap indicates mild overfitting, which is expected given the limited dataset size. The plateau in the validation curve suggests that adding more data may yield only marginal improvements unless combined with feature engineering or regularization adjustments.
- Confidence Intervals: The shaded regions show variability across folds. Wider intervals at smaller training sizes reflect instability due to limited data, while narrower intervals at larger sizes indicate more consistent performance.

After this, confusing matrixes were constructed to further evaluate the values predicted and the actual values of the test dataset.

3.5.1 Confusion Matrixes

```
# Define confusion matrices
conf_matrices = {}

# Tuned models
for name, model in tuned_models.items():
    y_pred = model.predict(X_test)
    cm = confusion_matrix(y_test, y_pred)
    conf_matrices[name] = cm

name_mapping = {
    'LogisticRegression_Tuned': 'Logistic Regression Tuned',
    'RandomForest_Tuned': 'Random Forest Tuned',
    'XGBoost_Tuned': 'XGBoost Tuned'
}

# Plot confusion matrices
for name, cm in conf_matrices.items():
    display_name = name_mapping.get(name, name)

    # Normalize for percentages
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] * 100

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    # Counts
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[0])
    axes[0].set_title(f'{display_name} - Counts')
    axes[0].set_xlabel('Predicted')
    axes[0].set_ylabel('Actual')

    # Percentages
    sns.heatmap(cm_normalized, annot=True, fmt='.2f', cmap='Greens', ax=axes[1])
    axes[1].set_title(f'{display_name} - Percentages')
    axes[1].set_xlabel('Predicted')
    axes[1].set_ylabel('Actual')

    plt.tight_layout()
    plt.show()
```

After this, confusing matrixes were constructed to further evaluate the values predicted and the actual values of the test dataset.



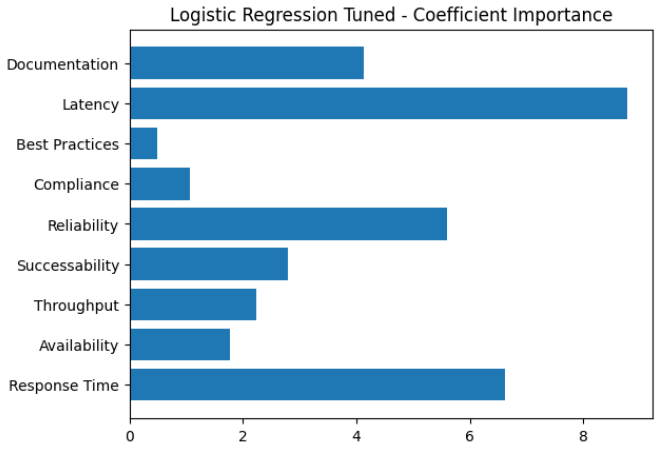
Each confusion matrix shows actual values vs predicted classes. The left matrix displays raw counts, while the right matrix shows percentages normalized per actual class. This helps identify which classes are predicted accurately and where misclassifications occur relative to class size.

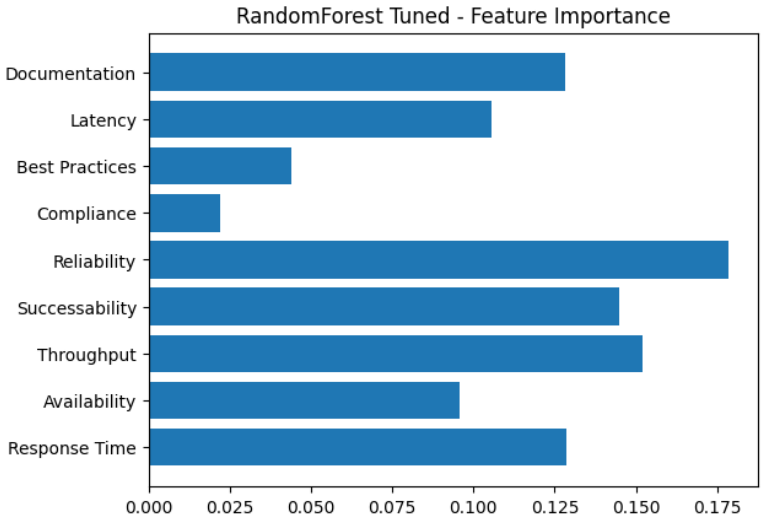
Following the same analysis of the previous type of models (regression), a feature importance analysis was made for all tuned classification models.

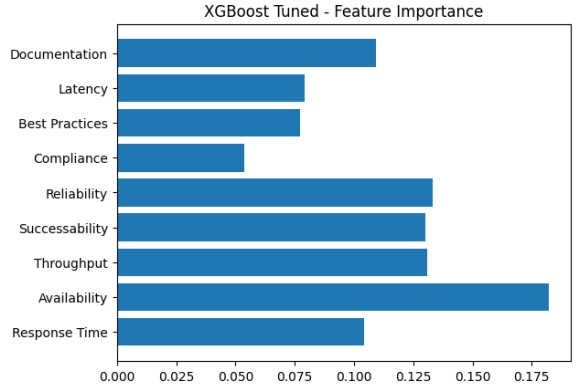
4. Interpretability

4.1 Feature Importance

```
# RandomForest Tuned
rf_model = random_search.best_estimator_
importances_rf = rf_model.feature_importances_
plt.barh(feature_cols, importances_rf)
plt.title("RandomForest Tuned - Feature Importance")
plt.show()

# XGBoost Tuned
xgb_model = xgb_search.best_estimator_
importances_xgb = xgb_model.feature_importances_
plt.barh(feature_cols, importances_xgb)
plt.title("XGBoost Tuned - Feature Importance")
plt.show()

# Logistic Regression Tuned
lr_model = lr_search.best_estimator_
importances_lr = np.mean(np.abs(lr_model.named_steps['classifier'].coef_), axis=0)
plt.barh(feature_cols, importances_lr)
plt.title("Logistic Regression Tuned - Coefficient Importance")
plt.show()
```







1) Random Forest Tuned Importance:

- Top Feature: Reliability is the most influential feature, indicating that service reliability strongly impacts the predicted quality level.

- Other High-Impact Features: Throughput and Successability follow closely, suggesting performance and success rate are critical.

- Moderate Importance: Documentation and Response Time still play a role but less than reliability.

- Least Important: Compliance and Best Practices have minimal impact, meaning they contribute little to prediction accuracy.

2) XGBoost Tuned Importance:

- Top Feature: Availability dominates, showing that uptime is the strongest predictor for XGBoost.

- Balanced Importance: Reliability, Successability, and Throughput are all significant, reinforcing the importance of performance metrics.

- Moderate Role: Documentation and Response Time are secondary factors.

- Least Important: Compliance and Best Practices remain low, similar to Random Forest.

3) Logistic Regression Importance:

- Top Feature: Latency has the highest coefficient magnitude, meaning delays strongly influence the predicted class.

- Second Strongest: Response Time also matters significantly.

- Moderate Impact: Reliability and Documentation contribute meaningfully.

- Low Impact: Best Practices and Compliance have very small coefficients, confirming their limited predictive power.

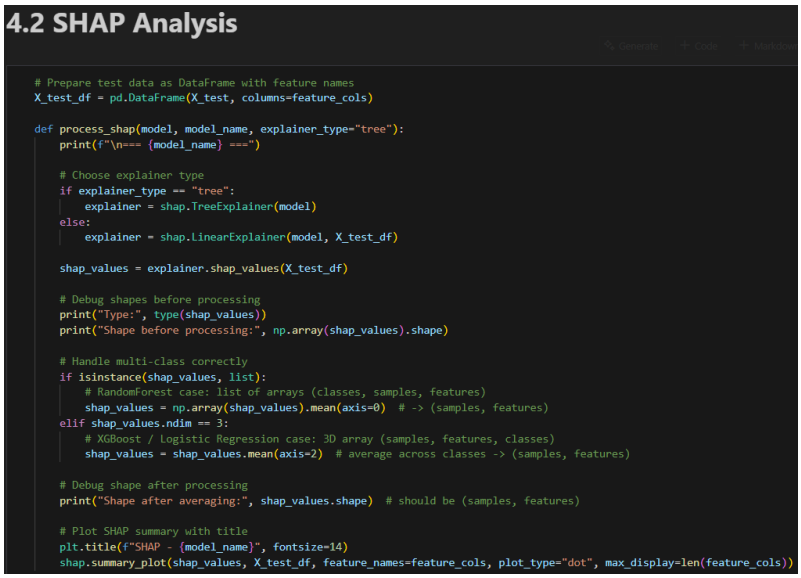
Overall:

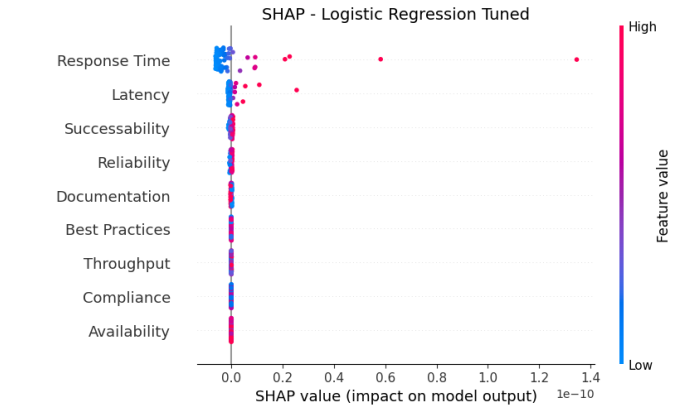
- Across all models, features tied to speed (Latency, Response Time) and stability (Reliability, Availability) consistently rank high.

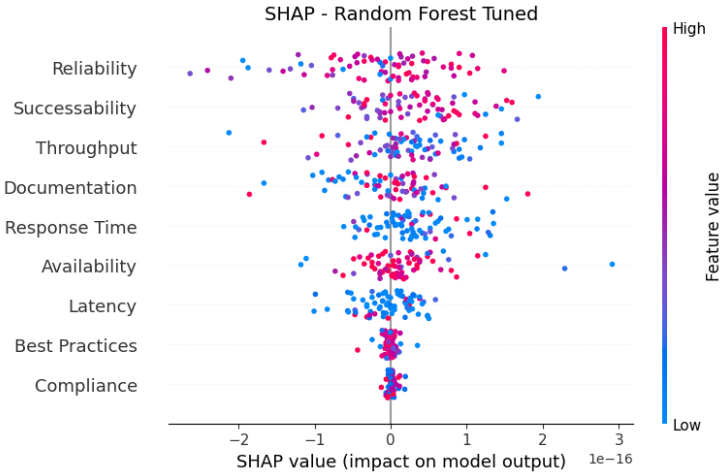
- Documentation is Secondary: While not negligible, documentation is less critical than technical performance.

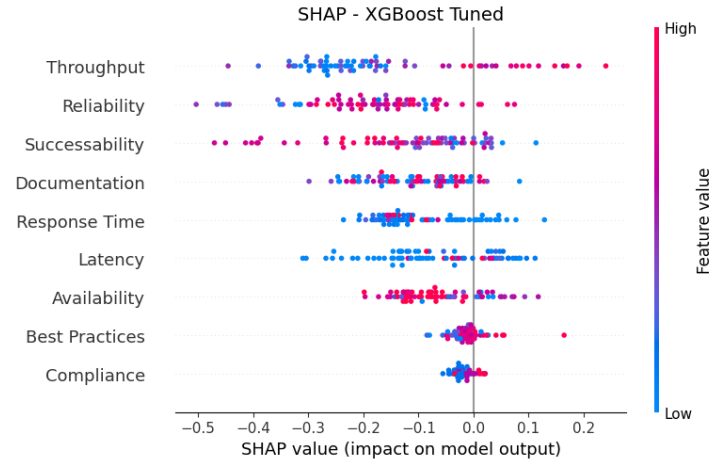
- Compliance & Best Practices Are Weak Predictors consistently rank lowest, suggesting they do not strongly differentiate service quality levels.

After this a SHAP graph was made for all tuned models as per identified in the following pictures.









Top Influential Features Across Models:

- Response Time and Successability consistently show the strongest impact on predictions.
- Throughput and Availability also contribute but to a lesser extent.

- Interaction Effects: SHAP interaction plots indicate weak feature interactions overall, meaning predictions rely on combined effects rather than a single dominant feature.

Model-Specific Observations:

- Random Forest and XGBoost emphasize performance and reliability metrics.
- Logistic Regression highlights latency-related features (e.g., Response Time).

Interpretability Insight: No single feature overwhelmingly drives predictions; instead, multiple features collectively influence the outcome, reinforcing the need for a balanced feature set.

To end the classification models for the first task, the best model was exported with all metrics into a json and csv file, and also a md file was constructed following the same procedure as the regression models.

5. Export Metrics

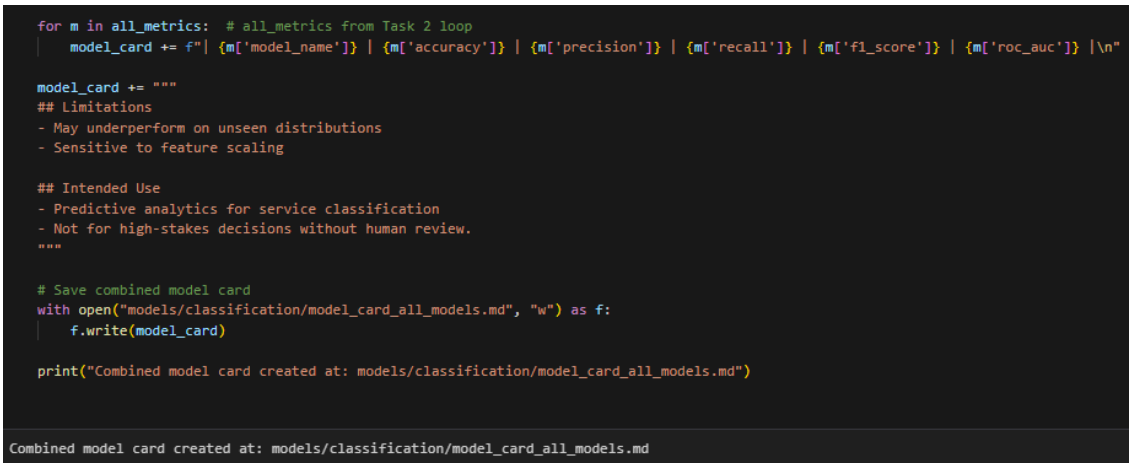
5.1 Save Model + Preprocessor

```
# 5.1 Save Best Model with Preprocessor
# Instead of using a standalone scaler, saves pipeline so preprocessing steps are included

package = {
    "model": best_model,          # Best tuned model (pipeline or estimator)
    "features": feature_cols,     # Feature names for reference
    "classes": label_encoder.classes_  # Original class labels
}

# Save the model package
model_path = f"models/classification/model_package_{best_model_name.replace(' ', '_')}.pkl"
dump(package, model_path)
print(f"Saved model package at: {model_path}")
```

5.2 Model Card

Markdown file summarizing the model

```
all_metrics = []  # Collect metrics for all tuned models
classes = np.unique(y_test)

for name, model in tuned_models.items():
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)
    metrics = {
        "model_name": name,
        "accuracy": round(accuracy_score(y_test, y_pred), 4),
        "precision": round(precision_score(y_test, y_pred, average='macro'), 4),
        "recall": round(recall_score(y_test, y_pred, average='macro'), 4),
        "f1_score": round(f1_score(y_test, y_pred, average='macro'), 4),
        "roc_auc": round(roc_auc_score(label_binarize(y_test, classes=classes), y_prob, average='macro', multi_class='ovr'), 4)
    }
    all_metrics.append(metrics)


# Build combined model card
model_card = f"""
# Model Card: Classification Models Comparison
**Version:** v1.0
**Date:** 2025-11-24
**Owner:** Data & AI Academy

## Data
- Source: QWS dataset (QoS metrics for web services)
- Target: Service Class (1-4)
- Preprocessing: Scaling, encoding

## Features
- {', '.join(feature_cols)}

## Models and Performance
| Model                | Accuracy | Precision | Recall | F1 Score | ROC-AUC |
|----------------------|----------|-----------|--------|----------|---------|
"""
```



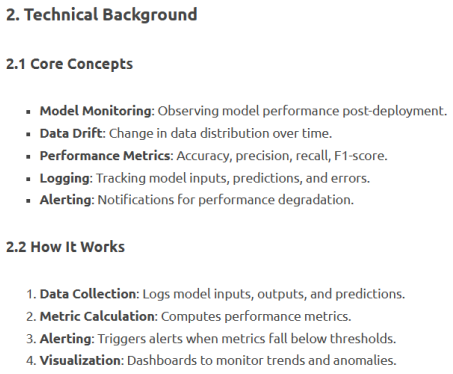
```
for m in all_metrics:  # all_metrics from Task 2 loop
    model_card += f"| {m['model_name']} | {m['accuracy']} | {m['precision']} | {m['recall']} | {m['f1_score']} | {m['roc_auc']} |\n"

model_card += """
## Limitations
- May underperform on unseen distributions
- Sensitive to feature scaling

## Intended Use
- Predictive analytics for service classification
- Not for high-stakes decisions without human review.
"""

# Save combined model card
with open("models/classification/model_card_all_models.md", "w") as f:
    f.write(model_card)

print("Combined model card created at: models/classification/model_card_all_models.md")
```

```
Combined model card created at: models/classification/model_card_all_models.md
```

Week 3

Investigation links:

1. [How to Build Model Monitoring Systems in Production: A Step-by-Step Guide | Codez Up]



2. Technical Background

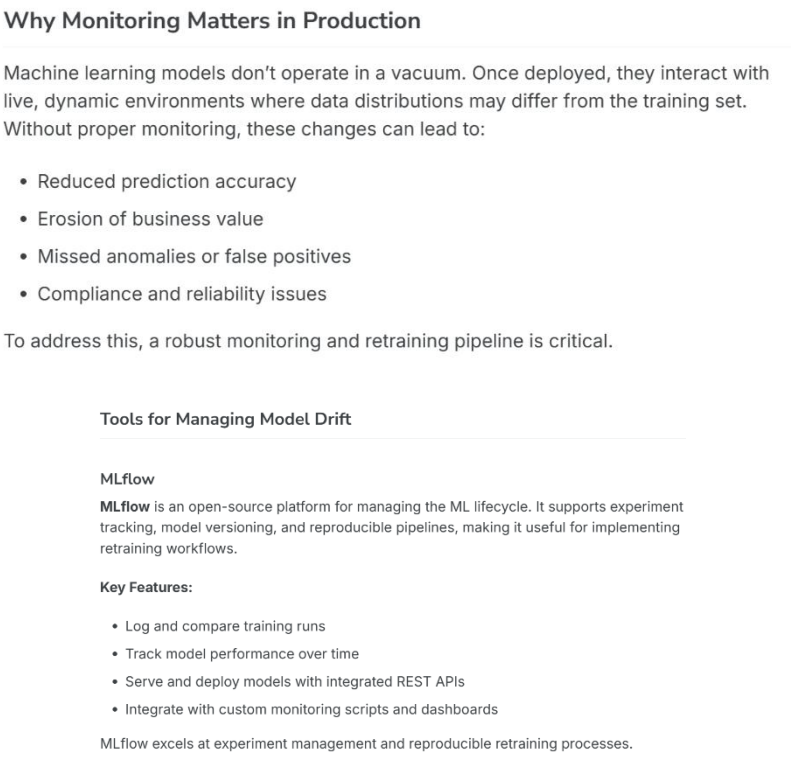
2.1 Core Concepts

- **Model Monitoring**: Observing model performance post-deployment.
- **Data Drift**: Change in data distribution over time.
- **Performance Metrics**: Accuracy, precision, recall, F1-score.
- **Logging**: Tracking model inputs, predictions, and errors.
- **Alerting**: Notifications for performance degradation.

2.2 How It Works

1. **Data Collection**: Logs model inputs, outputs, and predictions.
2. **Metric Calculation**: Computes performance metrics.
3. **Alerting**: Triggers alerts when metrics fall below thresholds.
4. **Visualization**: Dashboards to monitor trends and anomalies.

2. [Techniques for Monitoring and Managing Model Drift in Production - Data Science]



Why Monitoring Matters in Production

Machine learning models don't operate in a vacuum. Once deployed, they interact with live, dynamic environments where data distributions may differ from the training set. Without proper monitoring, these changes can lead to:

- Reduced prediction accuracy
- Erosion of business value
- Missed anomalies or false positives
- Compliance and reliability issues

To address this, a robust monitoring and retraining pipeline is critical.

Tools for Managing Model Drift

MLflow

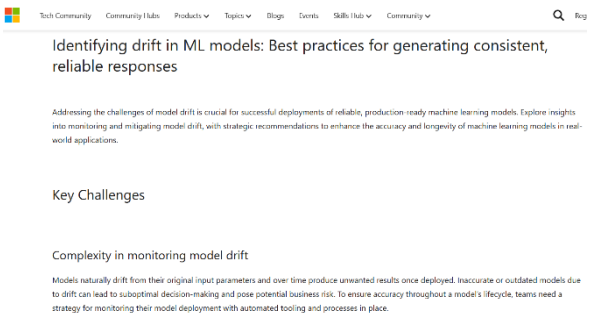
MLflow is an open-source platform for managing the ML lifecycle. It supports experiment tracking, model versioning, and reproducible pipelines, making it useful for implementing retraining workflows.

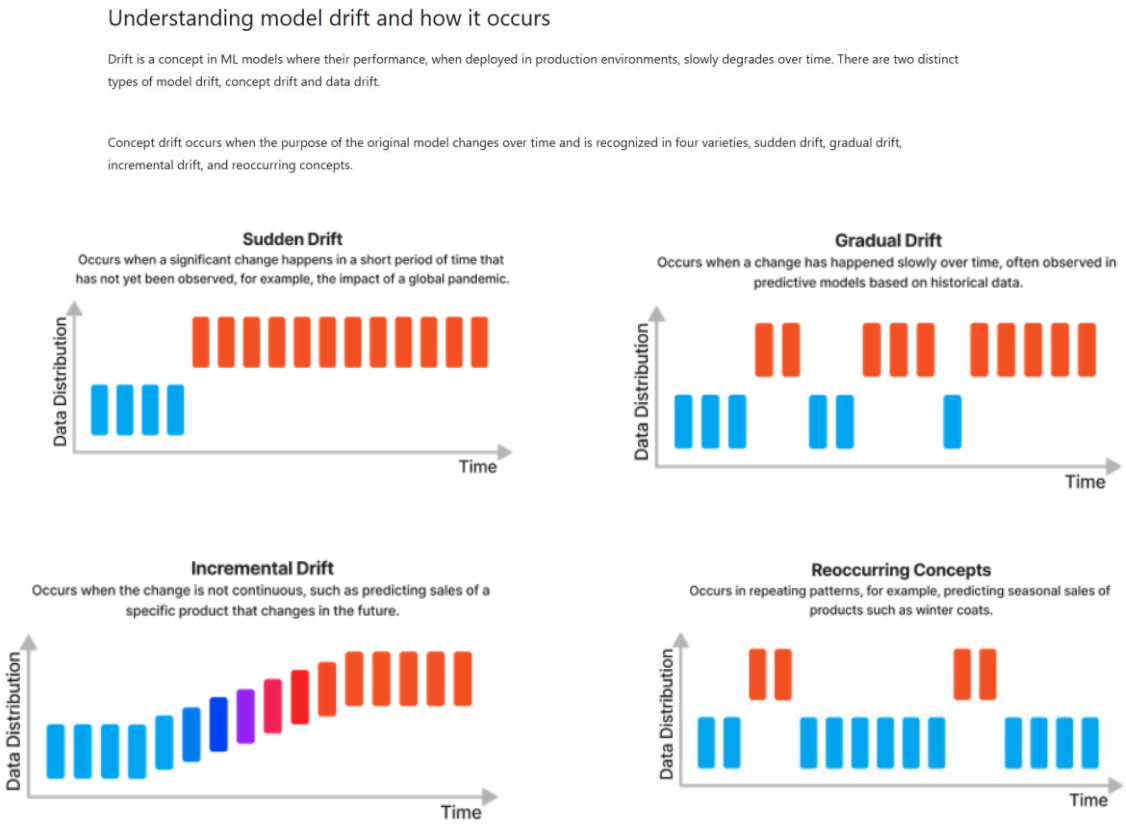
Key Features:

- Log and compare training runs
- Track model performance over time
- Serve and deploy models with integrated REST APIs
- Integrate with custom monitoring scripts and dashboards

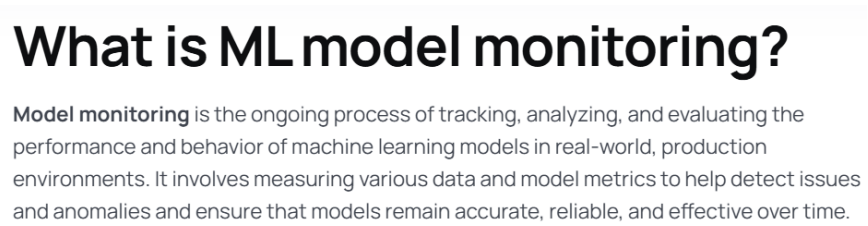
MLflow excels at experiment management and reproducible retraining processes.

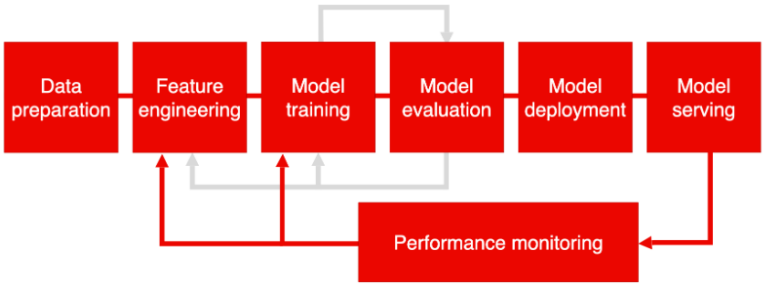
3. [Identifying drift in ML models: Best practices for generating consistent, reliable responses]



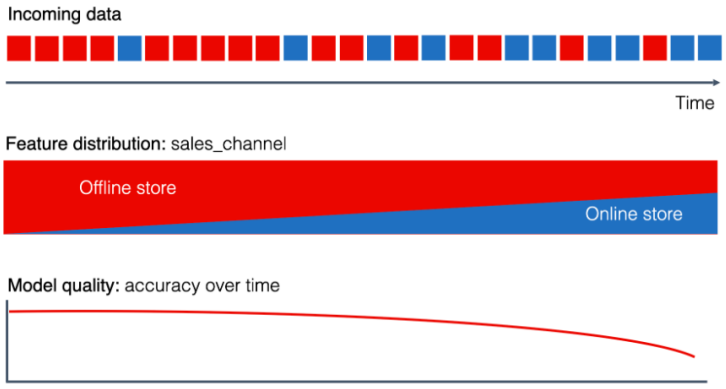


4. [Model monitoring for ML in production: a comprehensive guide]





Data drift. Data distribution drift occurs when the statistical properties of the input data change. An example could be a change in customer demographics, resulting in the model underperforming on a previously unseen customer segment.



Regression models

For the following task, it is necessary to do the following procedure:

- Define official metric to evaluate the model
- Implement k-fold or time-based back tests (in this case we will do k-fold back tests since the data doesn't have time base data)
- Generate plots for data analysis
- Data drift checks
- Report summary

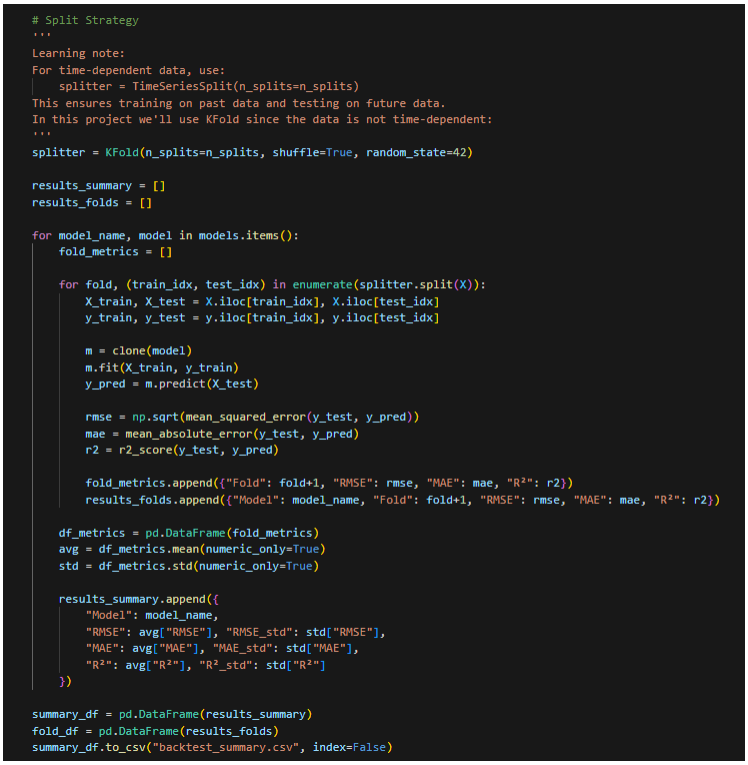
The regression models were evaluated first. In this case the defined metrics were R^2 for the primary metric and then for secondary metrics RMSE and MAE in order to give a more overall overview of the results. For this, the tuned models from the previous task were loaded into the script as the picture shows.

```
# Models
best_lr = joblib.load("../models/regression/best_lr_pipeline.pkl")
best_rf = joblib.load("../models/regression/best_rf_pipeline.pkl")
best_xgb = joblib.load("../models/regression/best_xgb_pipeline.pkl")

models = {
    "LinearRegression_Tuned": best_lr,
    "RandomForest_Tuned": best_rf,
    "XGBoost_Tuned": best_xgb
}

# Validation data
df = pd.read_csv("../qws1_dataset/validation_data_regression.csv")
target = "WsRF: Web Service Relevancy Function (%)"
X_val = df.drop(columns=[target])
y_val = df[target]
```

Then the back-test was started by dividing the validation data (data split from the original dataset that wasn't trained by the model or evaluated, in other words it's data completely new to the model). The idea is to divide the data into k (number) equal parts, train the model on the current k split of the data and test the model on the remaining folds. This process is iterative to finally receive an average performance across all k folds of data. The next figure shows the configuration steps prepared.

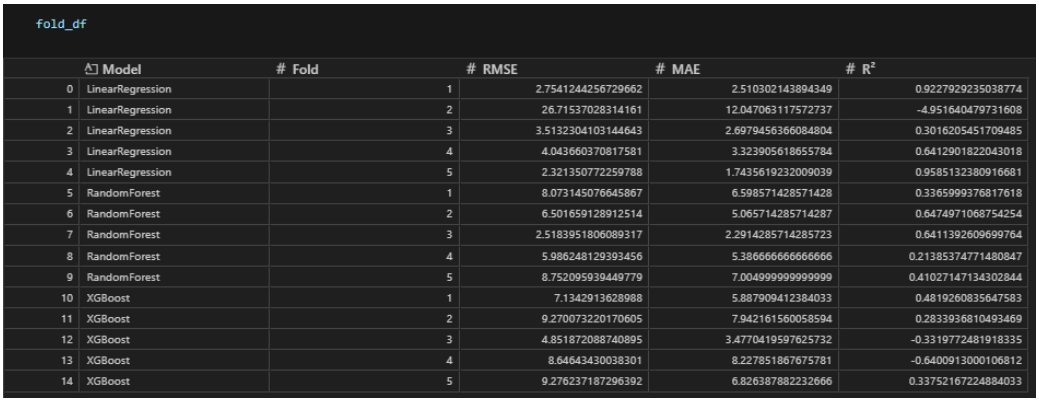
```
# Configuration
target = "WsRF: Web Service Relevancy Function (%)"
n_splits = 5

# Target Column clean
X = df.drop(columns=[target])
y = df[target]

# Models
models = {
    "LinearRegression": LinearRegression(),
    "RandomForest": RandomForestRegressor(random_state=42),
    "XGBoost": XGBRegressor(random_state=42)
}
```

Then finally, in the next figure, shows the split strategy for the k folds, and the code required in order to do the back test for the tuned regression models.



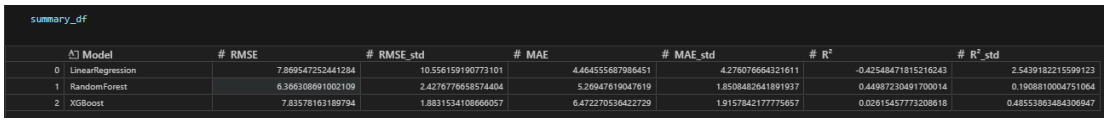
```python
# Split Strategy
'''
Learning note:
For time-dependent data, use:
    splitter = TimeSeriesSplit(n_splits=n_splits)
This ensures training on past data and testing on future data.
In this project we'll use KFold since the data is not time-dependent:
'''
splitter = KFold(n_splits=n_splits, shuffle=True, random_state=42)

results_summary = []
results_folds = []

for model_name, model in models.items():
    fold_metrics = []

    for fold, (train_idx, test_idx) in enumerate(splitter.split(X)):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

        m = clone(model)
        m.fit(X_train, y_train)
        y_pred = m.predict(X_test)

        rmse = np.sqrt(mean_squared_error(y_test, y_pred))
        mae = mean_absolute_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        fold_metrics.append({"Fold": fold+1, "RMSE": rmse, "MAE": mae, "R²": r2})
        results_folds.append({"Model": model_name, "Fold": fold+1, "RMSE": rmse, "MAE": mae, "R²": r2})

    df_metrics = pd.DataFrame(fold_metrics)
    avg = df_metrics.mean(numeric_only=True)
    std = df_metrics.std(numeric_only=True)

    results_summary.append({
        "Model": model_name,
        "RMSE": avg["RMSE"], "RMSE_std": std["RMSE"],
        "MAE": avg["MAE"], "MAE_std": std["MAE"],
        "R²": avg["R²"], "R²_std": std["R²"]
    })

summary_df = pd.DataFrame(results_summary)
fold_df = pd.DataFrame(results_folds)
summary_df.to_csv("backtest_summary.csv", index=False)
```

The next 2 figures show the results for each fold and the average for each metric decided previously.



```
fold_df
```

	Model	Fold	RMSE	MAE	R^2
0	LinearRegression	1	2.7541244256729662	2.510302143894349	0.9227929235038774
1	LinearRegression	2	26.71537028314161	12.047063117572737	-4.951640479731608
2	LinearRegression	3	3.5132304103144643	2.6979456366084804	0.3016205451709485
3	LinearRegression	4	4.043660370817581	3.323905618655784	0.6412901822043018
4	LinearRegression	5	2.321350772259788	1.7435619232009039	0.9585132380916681
5	RandomForest	1	8.073145076645867	6.598571428571428	0.3365999376817618
6	RandomForest	2	6.501659128912514	5.065714285714287	0.6474971068754254
7	RandomForest	3	2.5183951806089317	2.2914285714285723	0.6411392609699764
8	RandomForest	4	5.986248129393456	5.386666666666666	0.21385374771480847
9	RandomForest	5	8.752095939449779	7.004999999999999	0.41027147134302844
10	XGBoost	1	7.1342913628988	5.887909412384033	0.4819260835647583
11	XGBoost	2	9.270073220170605	7.942161560058594	0.2833936810493469
12	XGBoost	3	4.851872088740895	3.4770419597625732	-0.3319772481918335
13	XGBoost	4	8.64643430038301	8.227851867675781	-0.6400913000106812
14	XGBoost	5	9.276237187296392	6.826387882232666	0.33752167224884033



```
summary_df
```

	Model	RMSE	RMSE_std	MAE	MAE_std	R^2	R^2_std
0	LinearRegression	7.869547252441284	10.556159190773101	4.464555687986451	4.276076664321611	-0.42548471815216243	2.5439182215599123
1	RandomForest	6.366308691002109	2.4276776658574404	5.26947619047619	1.8508482641891937	0.44987230491700014	0.1908810004751064
2	XGBoost	7.83578163189794	1.8831534108666057	6.472270536422729	1.9157842177775657	0.02615457773208618	0.4855386348430694

In the first picture it shows results for the metrics of each fold resulting in the following aspects:

1. **Linear Regression**
    - RMSE: Ranges from 2.75 (Fold 1) to 26.75 (Fold 2), indicating extreme variability.
    - R^2: From 0.92 (good fit) to -4.95 (very poor fit).

- Interpretation: The model is highly unstable across folds. Negative R^2 values mean the model performs worse than a simple mean predictor in some folds. This suggests Linear Regression cannot capture the complexity of the data and is sensitive to data splits.

2. **Random Forest**
   - RMSE: Between 2.51 (Fold 3) and 8.75 (Fold 5), much more consistent than Linear Regression.
   - R^2: Positive across all folds (0.21–0.64), showing moderate predictive power.
   - Interpretation: Random Forest is robust and stable, handling variability better than Linear Regression. It adapts well to non-linear relationships.
3. **XGBoost**
   - RMSE: From 3.47 (Fold 3) to 9.27 (Fold 5), slightly higher than Random Forest in some folds.
   - R^2: Varies widely (0.48 in Fold 1 to -0.64 in Fold 4), including negative values.
   - Interpretation: XGBoost shows inconsistent performance. While it performs well in some folds, negative R^2 in others indicates overfitting or sensitivity to certain data partitions.

After the back-testing, some plots for data analysis were made to evaluate the model on unseen data (validation data). The next figures show the code produced to construct these graphs. First some configuration aspects, then the actual code for the plots.

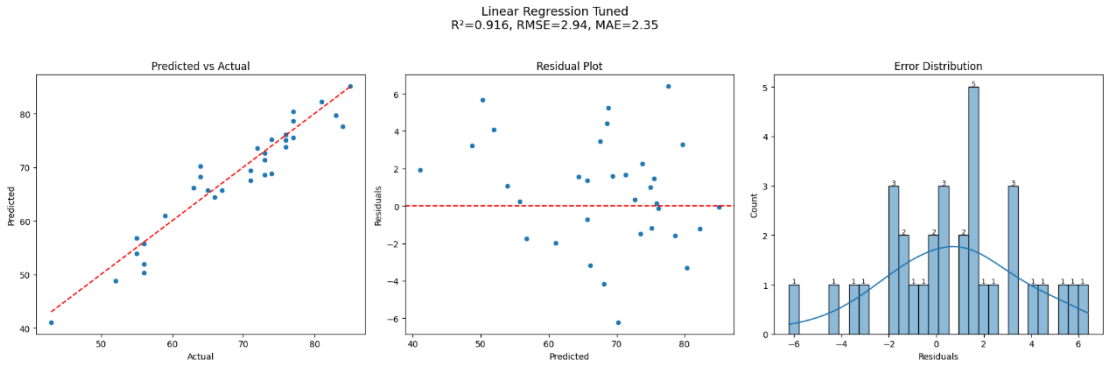
2. Configuration

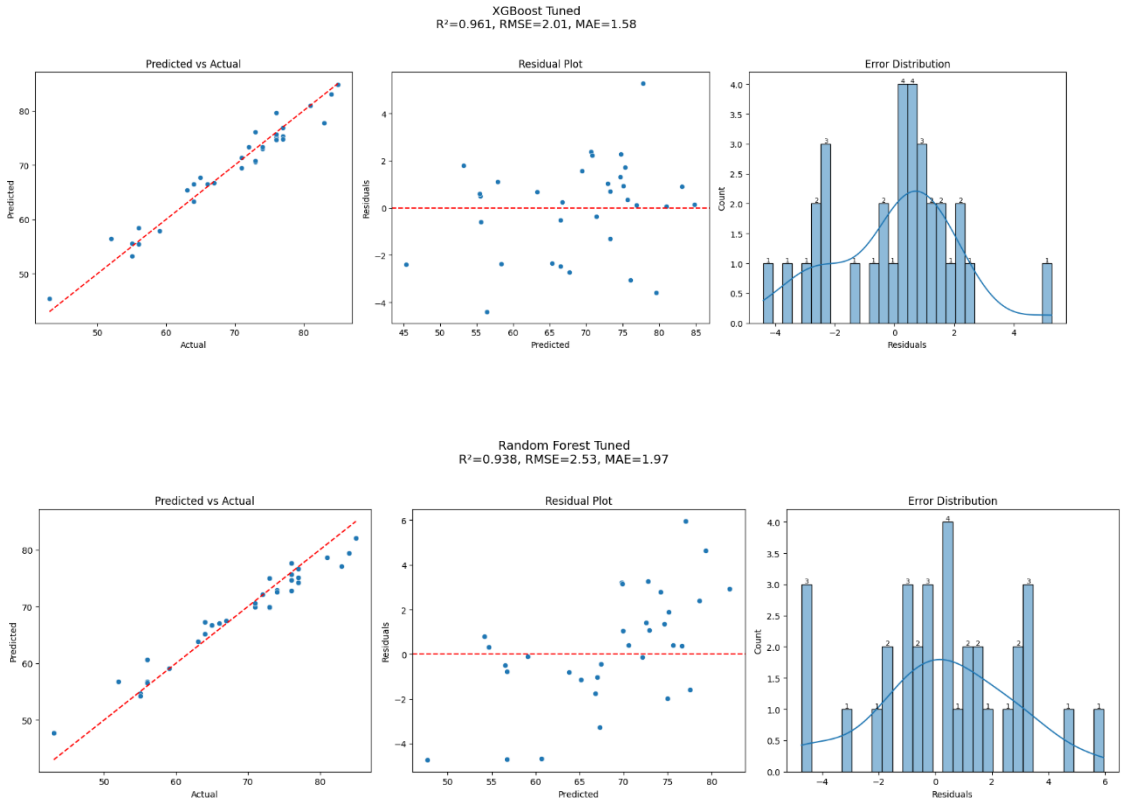
```
# Config
ARTIFACT_DIR = "artifacts_regression"
os.makedirs(ARTIFACT_DIR, exist_ok=True)

# Models
models = {
    "Linear Regression Tuned": joblib.load("../models/regression/best_lr_pipeline.pkl"),
    "Random Forest Tuned": joblib.load("../models/regression/best_rf_pipeline.pkl"),
    "XGBoost Tuned": joblib.load("../models/regression/best_xgb_pipeline.pkl")
}

# Validation data
df_val = pd.read_csv("../qws1_dataset/validation_data_regression.csv")
target = "WsRF: Web Service Relevancy Function (%)"
X_val = df_val.drop(columns=[target])
y_val = df_val[target]
```

3. Plotting

```python
for name, model in models.items():
    y_pred = model.predict(X_val)
    residuals = y_val - y_pred

    # Metrics
    rmse = np.sqrt(mean_squared_error(y_val, y_pred))
    mae = mean_absolute_error(y_val, y_pred)
    r2 = r2_score(y_val, y_pred)

    fig, axes = plt.subplots(1, 3, figsize=(18, 6))

    # Predicted vs Actual
    sns.scatterplot(x=y_val, y=y_pred, ax=axes[0])
    axes[0].plot([y_val.min(), y_val.max()], [y_val.min(), y_val.max()], 'r--')
    axes[0].set_title("Predicted vs Actual")
    axes[0].set_xlabel("Actual")
    axes[0].set_ylabel("Predicted")

    # Residual Plot
    sns.scatterplot(x=y_pred, y=residuals, ax=axes[1])
    axes[1].axhline(0, color='red', linestyle='--')
    axes[1].set_title("Residual Plot")
    axes[1].set_xlabel("Predicted")
    axes[1].set_ylabel("Residuals")

    # Error Distribution
    hist = sns.histplot(residuals, bins=30, kde=True, ax=axes[2])
    axes[2].set_title("Error Distribution")
    axes[2].set_xlabel("Residuals")

    for patch in hist.patches:
        height = patch.get_height()
        if height > 0:
            axes[2].text(
                patch.get_x() + patch.get_width() / 2,
                height,
                f"{int(height)}",
                ha="center", va="bottom", fontsize=8, color="black"
            )

    fig.suptitle(f"{name}\nR²={r2:.3f}, RMSE={rmse:.2f}, MAE={mae:.2f}", fontsize=14)

    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.savefig(f"{ARTIFACT_DIR}/{name}_plots.png")
    plt.show()
```

The previous figures produced the following graphs for each of the tuned ML models.





This results in the following analysis.

1. Linear Regression Tuned

- Predicted vs Actual Plot:
  - Points are close to the diagonal red line, indicating good alignment between predictions and actual values.
  - R^2 = 0.916 - The model explains 91.6% of variance.
  - Slight spread at higher values suggests minor under/over-prediction at extremes.
- Residual Plot
  - Residuals are scattered around zero without a clear pattern, no strong heteroscedasticity.
  - However, some residuals reach ±6, indicating occasional large errors.
- Error Distribution Plot
  - Histogram is roughly centered around zero but slightly skewed to the positive side.
  - Most residuals fall between -4 and +4, with a few outliers.
  - Shape is close to normal.

2. Random Forest Tuned:

- Predicted vs Actual Plot

- Points are tighter around the diagonal compared to Linear Regression.
- R^2 = 0.938, RMSE and MAE are lower - better performance than Linear Regression.
- Slight clustering near higher actual values, but overall strong fit.

- Residual Plot
  - Residuals are more evenly distributed than Linear Regression, but some negative residuals at lower predicted values.
  - No obvious trend, which is good.
- Error Distribution Plot
  - Centered near zero, slightly narrower spread than Linear Regression.
  - Fewer extreme residuals - Random Forest handles variability better.

3. XGBoost Tuned:

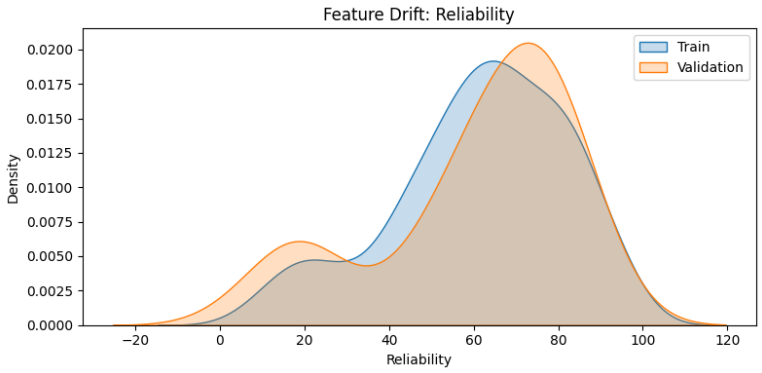
- Predicted vs Actual Plot
  - Points almost perfectly align with the diagonal.
  - R^2 = 0.961, RMSE = 2.01, MAE = 1.58 - best performance among all models.
  - Very tight clustering, minimal deviation.
- Residual Plot
  - Residuals are small and evenly scattered around zero.
  - No visible pattern.
- Error Distribution Plot:
  - Very narrow spread, most residuals between -3 and +3.
  - Almost symmetric around zero – strong indication of unbiased predictions.

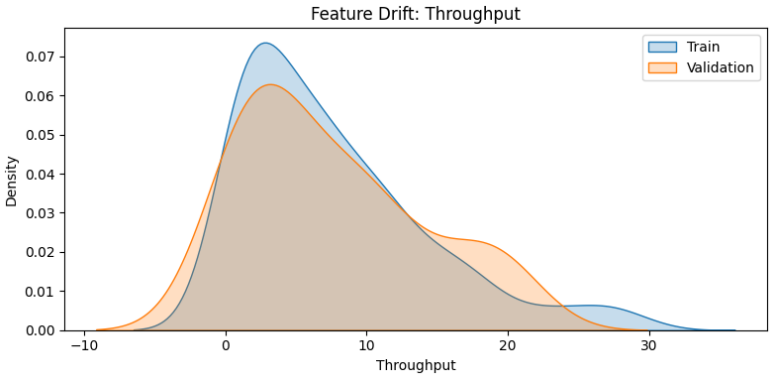
After this, an analysis of the data drift using the trained tuned models and the validation data to understand how the model acts on unseen data. The first figure shows the code to produce all the metrics necessary to conduct an analysis of this kind. The second figure shows the results in a table for organizational purposes. Just to clarify some information, the training data had 254 rows, validation data had 33 rows and for PSI thresholds – less than 0.1 (No drift), from 0.1 to 0.25 (Moderate drift) and more than 0.25 (Significant drift).

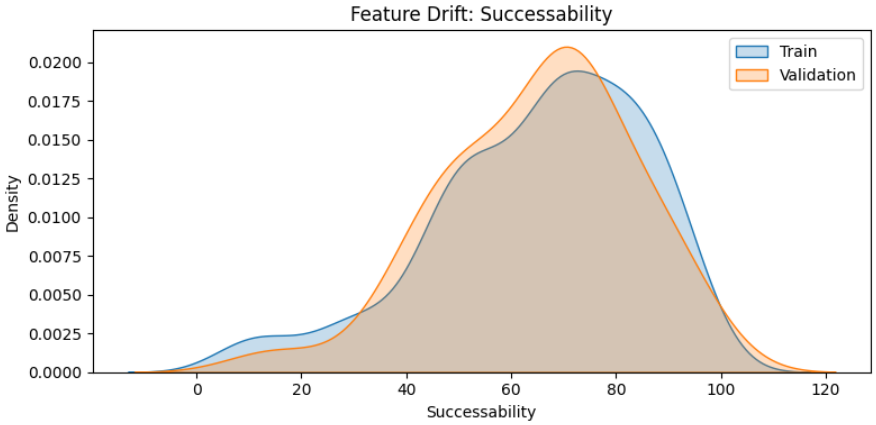
	Feature	PSI	PSI_Interpretation	KS_Statistic	KS_p_values
4	Reliability	1.9051663090330935	Significant drift	0.12264376043903603	0.7218697917931856
2	Throughput	0.9564699672042045	Significant drift	0.0688379861608208	0.9971193584934738
3	Successability	0.9406997529203982	Significant drift	0.10796945836315915	0.8454877830265335
1	Availability	0.598313821831578	Significant drift	0.0632307325220711	0.9991620200939377
6	Documentation	0.4402010701314691	Significant drift	0.20687186828919113	0.14060026660606562
0	Response Time	0.19005495841958076	Moderate drift	0.14626580768313052	0.5086471100503902
5	Compliance	0.06693674612053797	No drift	0.1096397041278931	0.8327112309513028

The top three features from, according to the PSI values are shown in the following figures.

```python
# Top 3 drifted features
top_features = df_drift['Feature'].head(3)
for col in top_features:
    plt.figure(figsize=(8, 4))
    sns.kdeplot(X_train[col], label='Train', fill=True)
    sns.kdeplot(X_val[col], label='Validation', fill=True)
    plt.title(f"Feature Drift: {col}")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{ARTIFACT_DIR}/drift_{col}.png")
    plt.show()
```







A. Reliability

- Validation distribution is shifted to the right compared to training.
- Indicates higher reliability values in validation data.
- PSI confirms this as the most drifted feature.

B. Throughput

- Validation curve shows heavier tail on the right.
- Suggests more high-throughput observations in validation compared to training.
- This could affect model predictions if throughput strongly influences the target.

C. Successability

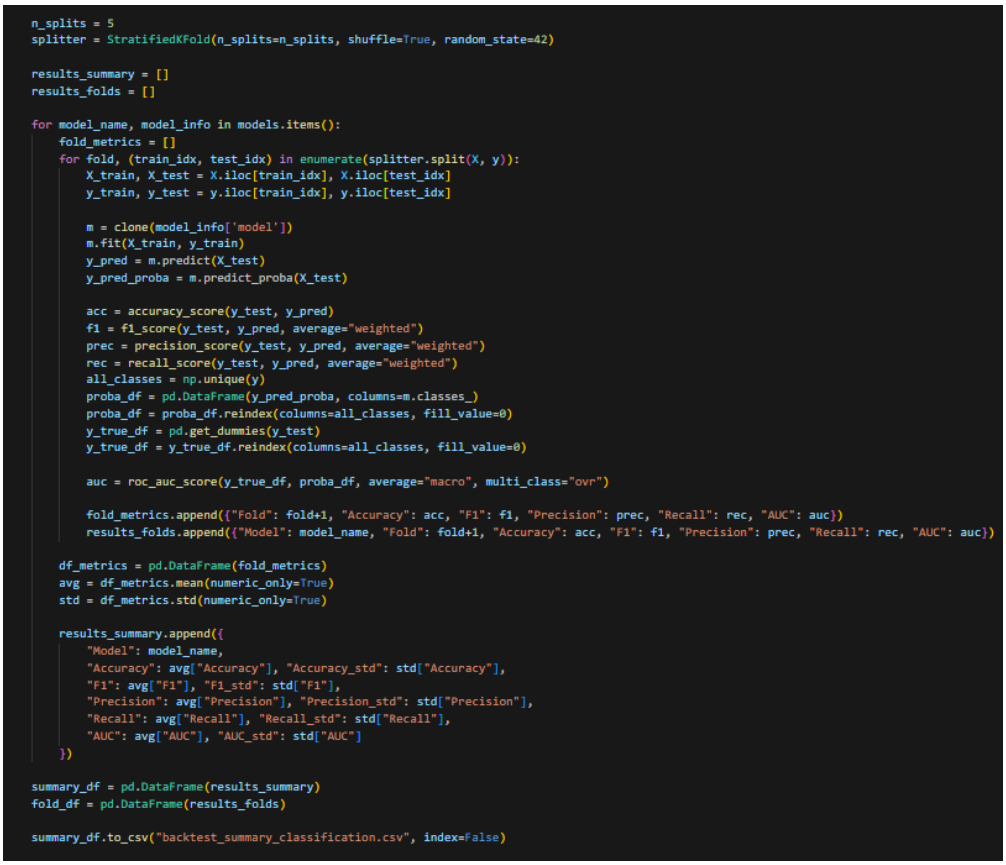
- Validation distribution slightly shifted right.
- Indicates improved successability in validation data.
- Drift is significant but less extreme than Reliability

All these results were saved for later use as part of the Route 25 project.

Classification models

A similar procedure was conducted for these types of ML models. The differences will be depicted in this document. As the first step, definition of metrics, the following are the metrics considered for evaluating - **Primary:** F1-score, **Secondary:** Accuracy, Precision, Recall, AUC – which will be evaluated for all 3 tuned models.

For the back-test, the next figures it is shown the code used for the test, the metrics for each fold (subset of the validation data). There is also another table that aggregates these values by the average for each mode after the figures.



```
n_splits = 5
splitter = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)

results_summary = []
results_folds = []

for model_name, model_info in models.items():
    fold_metrics = []
    for fold, (train_idx, test_idx) in enumerate(splitter.split(X, y)):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

        m = clone(model_info['model'])
        m.fit(X_train, y_train)
        y_pred = m.predict(X_test)
        y_pred_proba = m.predict_proba(X_test)

        acc = accuracy_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred, average="weighted")
        prec = precision_score(y_test, y_pred, average="weighted")
        rec = recall_score(y_test, y_pred, average="weighted")
        all_classes = np.unique(y)
        proba_df = pd.DataFrame(y_pred_proba, columns=m.classes_)
        proba_df = proba_df.reindex(columns=all_classes, fill_value=0)
        y_true_df = pd.get_dummies(y_test)
        y_true_df = y_true_df.reindex(columns=all_classes, fill_value=0)

        auc = roc_auc_score(y_true_df, proba_df, average="macro", multi_class="ovr")

        fold_metrics.append({"Fold": fold+1, "Accuracy": acc, "F1": f1, "Precision": prec, "Recall": rec, "AUC": auc})
        results_folds.append({"Model": model_name, "Fold": fold+1, "Accuracy": acc, "F1": f1, "Precision": prec, "Recall": rec, "AUC": auc})

    df_metrics = pd.DataFrame(fold_metrics)
    avg = df_metrics.mean(numeric_only=True)
    std = df_metrics.std(numeric_only=True)

    results_summary.append({
        "Model": model_name,
        "Accuracy": avg["Accuracy"], "Accuracy_std": std["Accuracy"],
        "F1": avg["F1"], "F1_std": std["F1"],
        "Precision": avg["Precision"], "Precision_std": std["Precision"],
        "Recall": avg["Recall"], "Recall_std": std["Recall"],
        "AUC": avg["AUC"], "AUC_std": std["AUC"]
    })

summary_df = pd.DataFrame(results_summary)
fold_df = pd.DataFrame(results_folds)

summary_df.to_csv("backtest_summary_classification.csv", index=False)
```

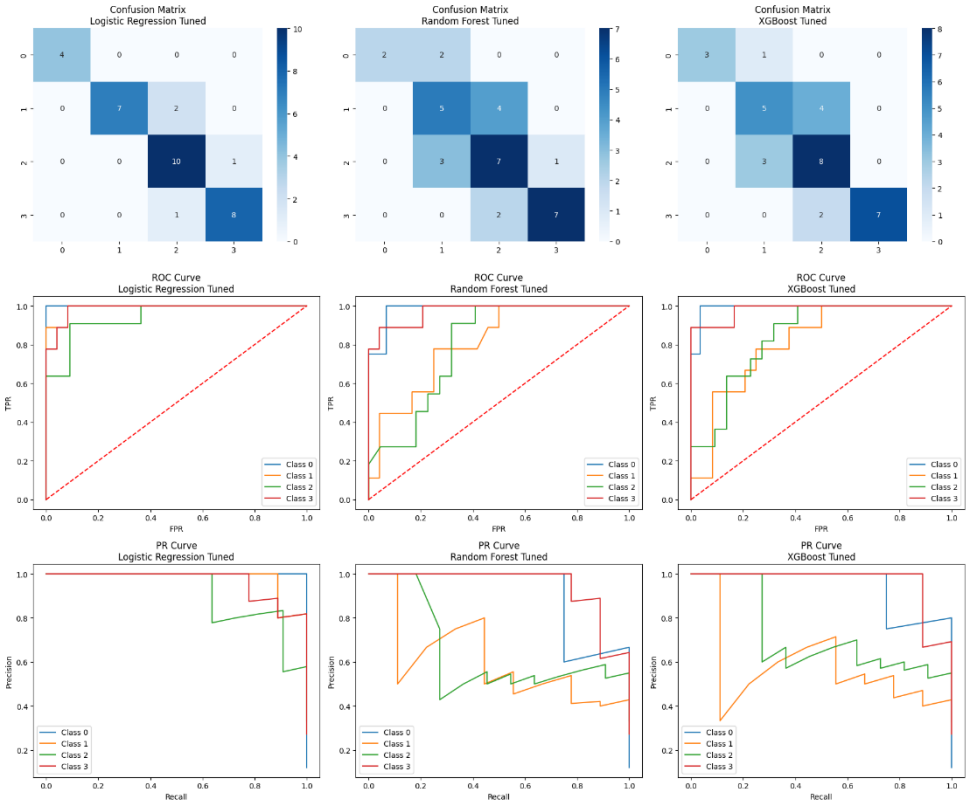
	Model	Fold	Accuracy	F1	Precision	Recall	AUC
0	LogisticRegression_Tuned	1	0.2857142857142857	0.23809523809523808	0.21428571428571427	0.2857142857142857	0.6916666666666667
1	LogisticRegression_Tuned	2	0.7142857142857143	0.7142857142857142	0.8571428571428571	0.7142857142857143	0.95
2	LogisticRegression_Tuned	3	0.42857142857142855	0.3333333333333333	0.2857142857142857	0.42857142857142855	0.6916666666666667
3	LogisticRegression_Tuned	4	0.8333333333333334	0.7777777777777777	0.75	0.8333333333333334	0.9
4	LogisticRegression_Tuned	5	0.5	0.43333333333333335	0.38888888888888884	0.5	*Missing value*
5	RandomForest_Tuned	1	0.2857142857142857	0.2857142857142857	0.2857142857142857	0.2857142857142857	0.625
6	RandomForest_Tuned	2	0.7142857142857143	0.7047619047619048	0.7619047619047619	0.7142857142857143	0.925
7	RandomForest_Tuned	3	0.5714285714285714	0.41904761904761906	0.3333333333333333	0.5714285714285714	0.8999999999999999
8	RandomForest_Tuned	4	0.6666666666666666	0.6388888888888888	0.7222222222222223	0.6666666666666666	0.9
9	RandomForest_Tuned	5	0.5	0.4666666666666666	0.4444444444444444	0.5	*Missing value*
10	XGBoost_Tuned	1	0.42857142857142855	0.38775510204081637	0.35714285714285715	0.42857142857142855	0.5166666666666666
11	XGBoost_Tuned	2	0.5714285714285714	0.5142857142857143	0.47619047619047616	0.5714285714285714	0.7583333333333333
12	XGBoost_Tuned	3	0.42857142857142855	0.3428571428571429	0.2857142857142857	0.42857142857142855	0.65
13	XGBoost_Tuned	4	0.5	0.48888888888888893	0.5555555555555555	0.5	0.86875
14	XGBoost_Tuned	5	0.16666666666666666	0.13333333333333333	0.1111111111111111	0.16666666666666666	*Missing value*

Model	Accuracy	F1	Precision	Recall	AUC
Logistic Regression	0,552381	0,499365	0,499206	0,552381	0,808333
Random Forest	0,547619	0,503016	0,509524	0,547619	0,8375
XGBoost	0,419048	0,373424	0,357143	0,419048	0,698437

After the back-testing some graphs were constructed to evaluate the tuned classification ML models, although in this case, it was built confusion matrices, ROC curves and PR (precision-recall) curves.

3. Plotting

```
classes = np.unique(y_val)
y_true_bin = label_binarize(y_val, classes=classes)

fig, axes = plt.subplots(3, len(models), figsize=(6 * len(models), 15))

for col, (name, model_info) in enumerate(models.items()):
    m = model_info["model"]
    y_pred_proba = m.predict_proba(X_val)
    y_pred = np.argmax(y_pred_proba, axis=1)

    # Confusion Matrix
    cm = confusion_matrix(y_val, y_pred)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", ax=axes[0, col])
    axes[0, col].set_title(f"Confusion Matrix\n{name}")

    # ROC Curve
    for i, cls in enumerate(classes):
        fpr, tpr, _ = roc_curve(y_true_bin[:, i], y_pred_proba[:, i])
        axes[1, col].plot(fpr, tpr, label=f"Class {cls}")
    axes[1, col].plot([0, 1], [0, 1], 'r--')
    axes[1, col].set_title(f"ROC Curve\n{name}")
    axes[1, col].set_xlabel("FPR")
    axes[1, col].set_ylabel("TPR")
    axes[1, col].legend()

    # Precision-Recall Curve
    for i, cls in enumerate(classes):
        prec_curve, rec_curve, _ = precision_recall_curve(y_true_bin[:, i], y_pred_proba[:, i])
        axes[2, col].plot(rec_curve, prec_curve, label=f"Class {cls}")
    axes[2, col].set_title(f"PR Curve\n{name}")
    axes[2, col].set_xlabel("Recall")
    axes[2, col].set_ylabel("Precision")
    axes[2, col].legend()

plt.tight_layout()
plt.savefig(f"{ARTIFACT_DIR}/combined_evaluation_plots.png")
plt.show()
```



1) **Confusion Matrices:**

- Logistic Regression Tuned
    - Most predictions are correct for classes 2 and 3.
    - Minor misclassifications occur between classes 0 and 1.
- Random Forest Tuned
    - Performs well on class 2 but shows more confusion between classes 0 and 1 compared to Logistic Regression.
    - Class 3 predictions are slightly less accurate than Logistic Regression.
- XGBoost Tuned
    - Similar to Random Forest, strong performance on class 2, but some misclassification between classes 0 and 1.
    - Overall balanced but slightly less precise than Logistic Regression for class 3.

Logistic Regression shows the most consistent diagonal dominance (better separation), while tree-based models have more overlap between lower classes.

2) **ROC Curves:**

All models show AUC values close to 1.0 for most classes, indicating strong discriminative ability.

- Logistic Regression: Curves are steep and close to the top-left corner for classes 2 and 3 - excellent performance.
- Random Forest & XGBoost: Slightly less steep for class 1, meaning weaker separation for that class compared to Logistic Regression.

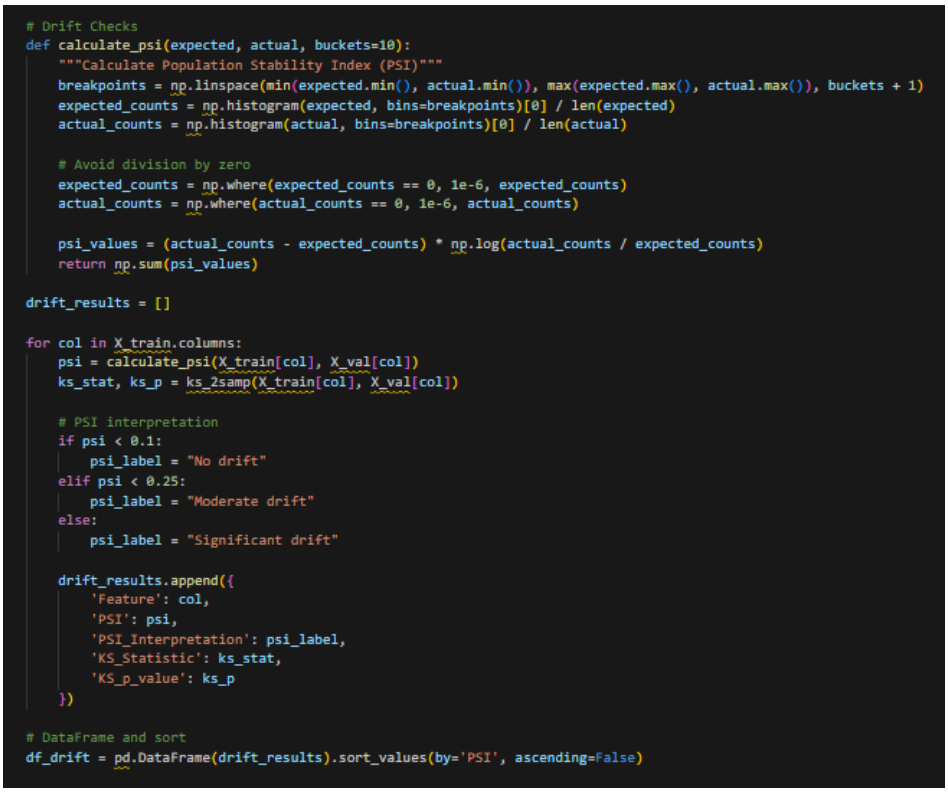
All models perform well overall, but Logistic Regression has the most consistent ROC curves across classes.

3) **Precision-Recall Curves**

- Logistic Regression: High precision across most recall levels for classes 2 and 3.
- Random Forest & XGBoost: More variability and lower precision for classes 0 and 1, especially at higher recall.

PR curves confirm that minority classes (likely class 0 or 1) are harder to predict accurately for tree-based models. Logistic Regression maintains better precision-recall trade-offs for dominant classes, while Random Forest and XGBoost struggle more with minority classes.

Following the same procedure as the regression models, a data drift analysis was conducted as shown in the following figures.



```python
# Drift Checks
def calculate_psi(expected, actual, buckets=10):
    """Calculate Population Stability Index (PSI)"""
    breakpoints = np.linspace(min(expected.min(), actual.min()), max(expected.max(), actual.max()), buckets + 1)
    expected_counts = np.histogram(expected, bins=breakpoints)[0] / len(expected)
    actual_counts = np.histogram(actual, bins=breakpoints)[0] / len(actual)

    # Avoid division by zero
    expected_counts = np.where(expected_counts == 0, 1e-6, expected_counts)
    actual_counts = np.where(actual_counts == 0, 1e-6, actual_counts)

    psi_values = (actual_counts - expected_counts) * np.log(actual_counts / expected_counts)
    return np.sum(psi_values)

drift_results = []

for col in X_train.columns:
    psi = calculate_psi(X_train[col], X_val[col])
    ks_stat, ks_p = ks_2samp(X_train[col], X_val[col])

    # PSI interpretation
    if psi < 0.1:
        psi_label = "No drift"
    elif psi < 0.25:
        psi_label = "Moderate drift"
    else:
        psi_label = "Significant drift"

    drift_results.append({
        'Feature': col,
        'PSI': psi,
        'PSI_Interpretation': psi_label,
        'KS_Statistic': ks_stat,
        'KS_p_value': ks_p
    })

# DataFrame and sort
df_drift = pd.DataFrame(drift_results).sort_values(by='PSI', ascending=False)
```

	Feature	PSI	PSI_Interpretation	KS_Statistic	KS_p_values
1	Availability	1.5599605667154053	Significant drift	0.11381531853972798	0.7987214170845921
3	Successability	1.451329624538141	Significant drift	0.11989978525411596	0.7460705818001309
2	Throughput	0.9370904939429537	Significant drift	0.09591982820329277	0.924028280160205
8	Documentation	0.8733962133134325	Significant drift	0.14077785731329037	0.5566375883551479
6	Best Practices	0.6161451334710712	Significant drift	0.08351228823669768	0.9748610313965934
0	Response Time	0.37885371752863656	Significant drift	0.14125507038892865	0.5524323931214813
7	Latency	0.254779475647047	Significant drift	0.10856597470770699	0.840510525790847
4	Reliability	0.25094177746941104	Significant drift	0.1380338821283703	0.581019520621139
5	Compliance	0.09938268317068456	No drift	0.05643044619422572	0.9998887054188534

From the results we can infer the following:

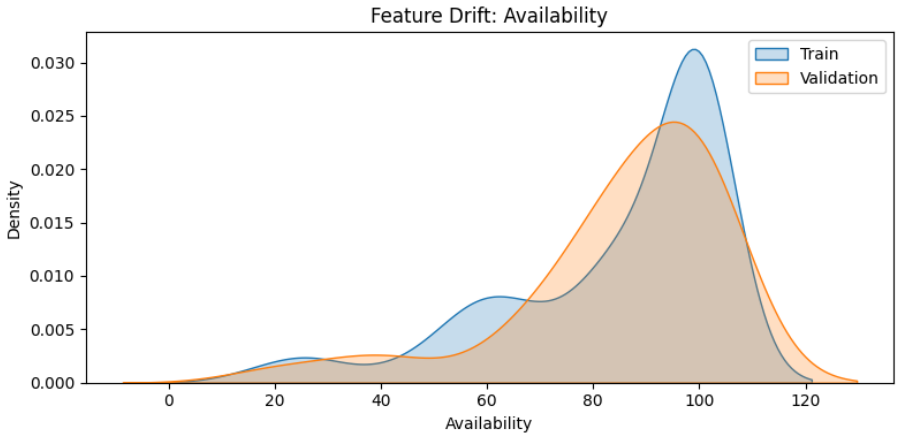
Availability (PSI = 1.56), Successability (1.45), Throughput (0.94), Documentation (0.87), Best Practices (0.61), and Response Time (0.38) show major distribution shifts between training and validation. Latency (0.25) and Reliability (0.25) are borderline significant drift.

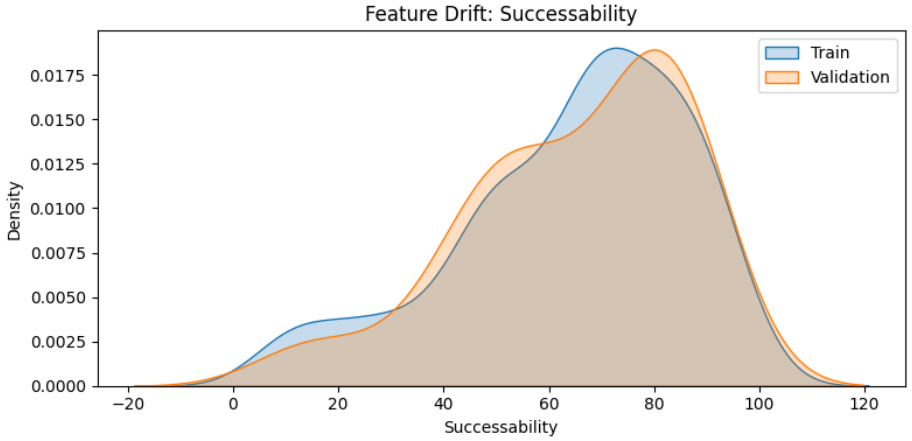
Stable Feature: Compliance (PSI = 0.09) shows no drift, its distribution remained consistent.

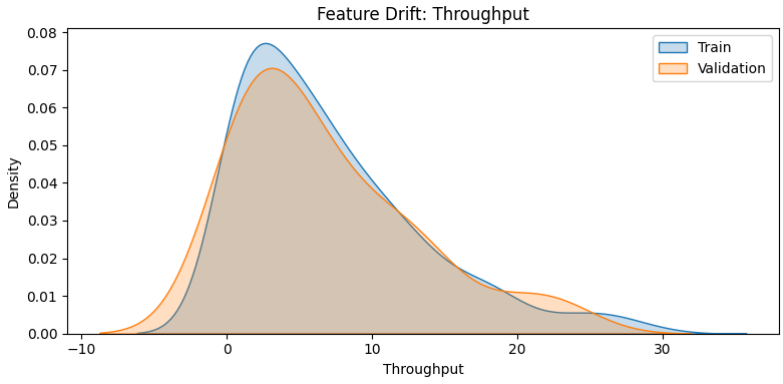
KS Test: KS p-values > 0.05 for all features -> shape differences are not statistically significant, but PSI highlights magnitude shifts.

Validation data differs substantially from training for key QoS features, which can affect model generalization. Monitoring or retraining may be needed.

```python
# 7. Plot top 3 drifted features
top_features = df_drift['Feature'].head(3)
for col in top_features:
    plt.figure(figsize=(8, 4))
    sns.kdeplot(X_train[col], label='Train', fill=True)
    sns.kdeplot(X_val[col], label='Validation', fill=True)
    plt.title(f"Feature Drift: {col}")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{ARTIFACT_DIR}/drift_{col}.png")
    plt.show()
```







For the three top features, according to the PSI value, here are the following analysis:

- Availability:
    - Validation curve shifted right (higher values) compared to training.
    - Indicates services in validation set have better availability than training.
    - Matches PSI = 1.56 (largest drift).
- Successability:
    - Slight right shift for validation curve.
    - Suggests improved successability in validation data.
    - PSI confirms significant drift (1.45).
- Throughput:
    - Validation curve shows heavier tail on the right.
    - Indicates more high-throughput observations in validation.
    - PSI = 0.94 -> strong drift.

About Capgemini Engineering

World leader in engineering and R&D services, Capgemini Engineering combines its broad industry knowledge and cutting-edge technologies in digital and software to support the convergence of the physical and digital worlds. Coupled with the capabilities of the rest of the Group, it helps clients to accelerate their journey towards Intelligent Industry. Capgemini Engineering has 65,000 engineer and scientist team members in over 30 countries across sectors including Aeronautics, Space, Defense, Naval, Automotive, Rail, Infrastructure & Transportation, Energy, Utilities & Chemicals, Life Sciences, Communications, Semiconductor & Electronics, Industrial & Consumer, Software & Internet.

Capgemini Engineering is an integral part of the Capgemini Group, a global business and technology transformation partner, helping organizations to accelerate their dual transition to a digital and sustainable world, while creating tangible impact for enterprises and society. It is a responsible and diverse group of 340,000 team members in more than 50 countries. With its strong over 55-year heritage, Capgemini is trusted by its clients to unlock the value of technology to address the entire breadth of their business needs. It delivers end-to-end services and solutions leveraging strengths from strategy and design to engineering, all fueled by its market leading capabilities in AI, cloud and data, combined with its deep industry expertise and partner ecosystem. The Group reported 2023 global revenues of €22.5 billion.

Get the future you want | www.capgemini.com