

Predictive Maintenance Dataset EDA and Modeling

This notebook performs an extensive exploratory data analysis (EDA) and model training on the Predictive Maintenance Dataset (AI4I 2020). It includes data cleaning, statistical summaries, visualizations, and the implementation of various machine learning models with hyperparameter tuning.

```
# =====
# 1. IMPORTS
# =====
import os
import seaborn as sns
import kagglehub
from sklearn.preprocessing import StandardScaler
from IPython.display import display
from sklearn.model_selection import train_test_split,
RepeatedStratifiedKFold, GridSearchCV
from sklearn.metrics import f1_score, classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from imblearn.over_sampling import SMOTE
import xgboost as xgb
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import warnings

warnings.filterwarnings("ignore")
sns.set_theme(style="white")
```

2-4. Preliminary to EDA

We begin by downloading the dataset and performing initial transformations to prepare it for analysis:

- **Loading:** Using `kagglehub` to fetch the AI4I 2020 sensor data.
- **Normalization:** Standardizing column names to a "snake_case" format and removing special characters (like brackets for units) to simplify programmatic access.
- **Feature Derivation:** Creating a `quality` feature (High, Medium, Low) from the `product_id` prefix, as machine failure rates often vary by product specification.
- **Dimensionality Reduction:** Dropping high-cardinality identifiers like `udi` and `product_id` that do not provide predictive value.

```

# =====
# 2. LOAD DATA
# =====
path = kagglehub.dataset_download("stephanmatzka/predictive-
maintenance-dataset-ai4i-2020")
csv_file_path = os.path.join(path, "ai4i2020.csv")
df = pd.read_csv(csv_file_path)

print("Dataset shape:", df.shape)
print(df.head())

Dataset shape: (10000, 14)
   UDI Product ID Type Air temperature [K] Process temperature
[K] \
0    1      M14860    M                298.1                 308.6
1    2      L47181    L                298.2                 308.7
2    3      L47182    L                298.1                 308.5
3    4      L47183    L                298.2                 308.6
4    5      L47184    L                298.2                 308.7

   Rotational speed [rpm]  Torque [Nm]  Tool wear [min]  Machine
failure TWF \
0           1551        42.8          0
0           0
1           1408        46.3          3
0           0
2           1498        49.4          5
0           0
3           1433        39.5          7
0           0
4           1408        40.0          9
0           0

   HDF  PWF  OSF  RNF
0    0    0    0    0
1    0    0    0    0
2    0    0    0    0
3    0    0    0    0
4    0    0    0    0

# =====
# 3. CLEAN COLUMN NAMES
# =====
df_eda = df.copy()

df_eda.columns = (

```

```

df_eda.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", "_")
    .str.replace("[", "_")
    .str.replace("]", "_")
)

# =====
# 4. FEATURE MODIFICATION
# =====
df_eda["quality"] = df_eda["product_id"].str[0].astype("category")

df_eda.drop(columns=["udi", "product_id"], inplace=True)
df_eda.drop(columns=["type"], inplace=True)

```

5. Defining the Problem

We establish the target variables for two distinct modeling tasks:

1. **Binary Classification:** Predicting the general `machine_failure` event.
2. **Multi-label Classification:** Predicting specific failure modes (`twf`, `hdf`, `pwf`, `osf`, `rnf`) to diagnose *why* a machine might fail.

```

# =====
# 5. TARGETS & FEATURES
# =====
binary_target = "machine_failure"
failure_modes = ["twf", "hdf", "pwf", "osf", "rnf"]

num_features = [
    "air_temperature_k",
    "process_temperature_k",
    "rotational_speed_rpm",
    "torque_nm",
    "tool_wear_min"
]

```

6-10. Integrity Checks and Statistical Summary

Before visualizing, we perform a rigorous audit of the data's health:

- **Data Quality:** Checking for missing values to ensure the dataset is complete.
- **Imbalance Detection:** Quantifying the "Failure Rate." Since failures are rare (~3.4%), this informs our need for specialized metrics (F1-score) and cost-sensitive learning later.
- **Segmentation:** Generating summary statistics (mean, std, median) grouped by failure status and product quality to identify early signals—such as whether higher torque or tool wear correlates with increased failure risk.

```

# =====
# 6. DATA CHECKS
# =====
print("\nMissing values:")
print(df_eda.isna().sum())

print("\nFailure distribution:")
print(df_eda[binary_target].value_counts(normalize=True))

print("\nFailure mode counts:")
print(df_eda[failure_modes].sum())
# =====
# 7. SUMMARY STATISTICS
# =====
summary_global = df_eda[num_features].describe().T
print("\nGlobal summary statistics:")
print(summary_global)
# =====
# 8. SUMMARY BY MACHINE FAILURE
# =====
summary_by_failure = (
    df_eda
    .groupby(binary_target)[num_features]
    .agg(["mean", "std", "min", "median", "max"])
)
print("\nSummary statistics by machine failure:")
print(summary_by_failure)
# =====
# 9. SUMMARY BY FAILURE MODE
# =====
for mode in failure_modes:
    print(f"\nSummary statistics for {mode.upper()} = 1")
    print(
        df_eda[df_eda[mode] == 1][num_features]
        .agg(["mean", "std", "min", "median", "max"])
    )
# =====
# 10. SUMMARY BY PRODUCT QUALITY
# =====
summary_by_quality = (
    df_eda
    .groupby("quality")[num_features]
    .agg(["mean", "std", "min", "median", "max"])
)
print("\nSummary statistics by product quality:")
print(summary_by_quality)

```

```

Missing values:
air_temperature_k      0
process_temperature_k   0
rotational_speed_rpm    0
torque_nm               0
tool_wear_min           0
machine_failure          0
twf                      0
hdf                      0
pwf                      0
osf                      0
rnf                      0
quality                  0
dtype: int64

Failure distribution:
machine_failure
0    0.9661
1    0.0339
Name: proportion, dtype: float64

Failure mode counts:
twf      46
hdf     115
pwf      95
osf      98
rnf      19
dtype: int64

Global summary statistics:
              count      mean       std      min     25%
\ air_temperature_k  10000.0  300.00493  2.000259  295.3  298.3
  process_temperature_k  10000.0  310.00556  1.483734  305.7  308.8
  rotational_speed_rpm  10000.0 1538.77610  179.284096 1168.0 1423.0
  torque_nm             10000.0   39.98691   9.968934    3.8   33.2
  tool_wear_min          10000.0  107.95100   63.654147    0.0   53.0

              50%      75%      max
air_temperature_k    300.1    301.5    304.5
process_temperature_k 310.1    311.1    313.8
rotational_speed_rpm 1503.0   1612.0   2886.0
torque_nm             40.1     46.8     76.6
tool_wear_min          108.0   162.0   253.0

```

```

Summary statistics by machine failure:
    air_temperature_k
        mean      std   min median   max \
machine_failure
0           299.973999  1.990748 295.3 300.0 304.5
1           300.886431  2.071473 295.6 301.6 304.4

process_temperature_k
    ... \
        mean      std   min median
max ...
machine_failure
...
0           309.995570  1.486846 305.7 310.0
313.8 ...
1           310.290265  1.363686 306.1 310.4
313.7 ...

    torque_nm          tool_wear_min
\       mean      std   min median   max      mean
machine_failure
0           39.629655  9.472080 12.6 39.9 70.0 106.693717
1           50.168142 16.374498  3.8 53.7 76.6 143.781711

    std min median   max
machine_failure
0           62.945790  0 107.0 246
1           72.759876  0 165.0 253

[2 rows x 25 columns]

Summary statistics for TWF = 1
    air_temperature_k process_temperature_k rotational_speed_rpm \
mean           300.297826            310.165217            1566.173913
std            1.945992             1.484021            205.768511
min           296.900000            307.400000            1323.000000
median         300.450000            310.250000            1516.000000
max           304.400000            313.700000            2271.000000

```

	torque_nm	tool_wear_min
mean	37.836957	216.369565
std	10.277232	12.257168
min	16.200000	198.000000
median	37.750000	214.500000
max	65.300000	253.000000

Summary statistics for HDF = 1

	air_temperature_k	process_temperature_k	rotational_speed_rpm
\mean	302.560870	310.788696	1337.260870
std	0.601853	0.644513	34.745967
min	300.800000	309.400000	1212.000000
median	302.500000	310.700000	1346.000000
max	303.700000	312.200000	1379.000000

	torque_nm	tool_wear_min
mean	53.166957	107.191304
std	6.223494	63.629257
min	41.600000	2.000000
median	52.600000	106.000000
max	68.200000	229.000000

Summary statistics for PWF = 1

	air_temperature_k	process_temperature_k	rotational_speed_rpm
\mean	300.075789	309.954737	1763.968421
std	2.147127	1.600450	620.829138
min	295.700000	306.200000	1200.000000
median	300.400000	310.200000	1386.000000
max	304.000000	313.200000	2886.000000

	torque_nm	tool_wear_min
mean	48.514737	101.884211
std	26.788653	64.355704
min	3.800000	0.000000
median	63.600000	100.000000
max	76.600000	234.000000

Summary statistics for OSF = 1

	air_temperature_k	process_temperature_k	rotational_speed_rpm
\mean	300.044898	310.073469	1350.326531
std	2.028565	1.511028	61.250841
min	295.600000	306.100000	1181.000000
median	300.050000	310.100000	1360.500000
max	304.000000	313.100000	1515.000000

	torque_nm	tool_wear_min
mean	58.370408	207.693878
std	5.943587	15.811002
min	46.300000	172.000000
median	57.700000	207.000000
max	75.400000	253.000000

Summary statistics for RNF = 1

	air_temperature_k	process_temperature_k	rotational_speed_rpm
\mean	300.815789	310.763158	1485.000000
std	1.707585	1.459532	109.438973
min	297.000000	307.700000	1306.000000
median	300.800000	311.000000	1481.000000
max	302.900000	312.500000	1687.000000

	torque_nm	tool_wear_min
mean	43.673684	124.473684
std	10.575908	69.550436
min	27.700000	2.000000
median	45.700000	144.000000
max	61.200000	215.000000

Summary statistics by product quality:

	air_temperature_k	mean	std	min	median	max
quality						
H	299.866999	2.021831	295.5	299.8	304.2	
L	300.015833	1.987453	295.3	300.1	304.5	
M	300.029263	2.017358	295.3	300.1	304.4	

	process_temperature_k	...
torque_nm		

		mean	std	min	median	max	...
mean							
quality							...
H		309.925723	1.489362	305.9	309.9	313.5	...
39.838285							
L		310.012300	1.475247	305.7	310.1	313.8	...
39.996600							
M		310.018785	1.498407	305.7	310.0	313.8	...
40.017251							
tool_wear_min							
		std	min	median	max	mean	std min
median	max						
quality							
H		9.642339	12.8	40.3	72.8	107.419741	63.080140 0
107.0	246						
L		10.012335	3.8	40.0	76.6	108.378833	64.058238 0
109.0	251						
M		9.992153	9.7	40.2	76.2	107.272272	63.044646 0
106.0	253						
[3 rows x 25 columns]							

11-14. Exploratory Data Analysis (EDA)

Visual exploration helps validate our statistical assumptions:

- **Distribution Analysis:** Histograms check for normality and outliers in sensor readings.
- **Conditional Density:** KDE plots visualize how numerical features overlap between "Pass" and "Fail" states, highlighting which sensors are the strongest predictors.
- **Correlation Mapping:** A correlation matrix identifies multi-collinearity (e.g., between temperatures) and quantifies the direct linear relationship between features and the failure target.

```
# =====
# 11. UNIVARIATE EDA
# =====
df_eda[num_features].hist(bins=30, figsize=(14, 8))
plt.suptitle("Numerical Feature Distributions")
plt.show()

# =====
# 12. FAILURE VS NON-FAILURE
# =====
for col in num_features:
    plt.figure(figsize=(6,4))
```

```

sns.kdeplot(
    data=df_eda,
    x=col,
    hue=binary_target,
    fill=True,
    common_norm=False
)
plt.title(f"{col} vs Machine Failure")
plt.show()

# =====
# 13. QUALITY VS FAILURE
# =====
print("\nFailure rate by quality:")
print(pd.crosstab(df_eda["quality"], df_eda[binary_target],
normalize="index"))

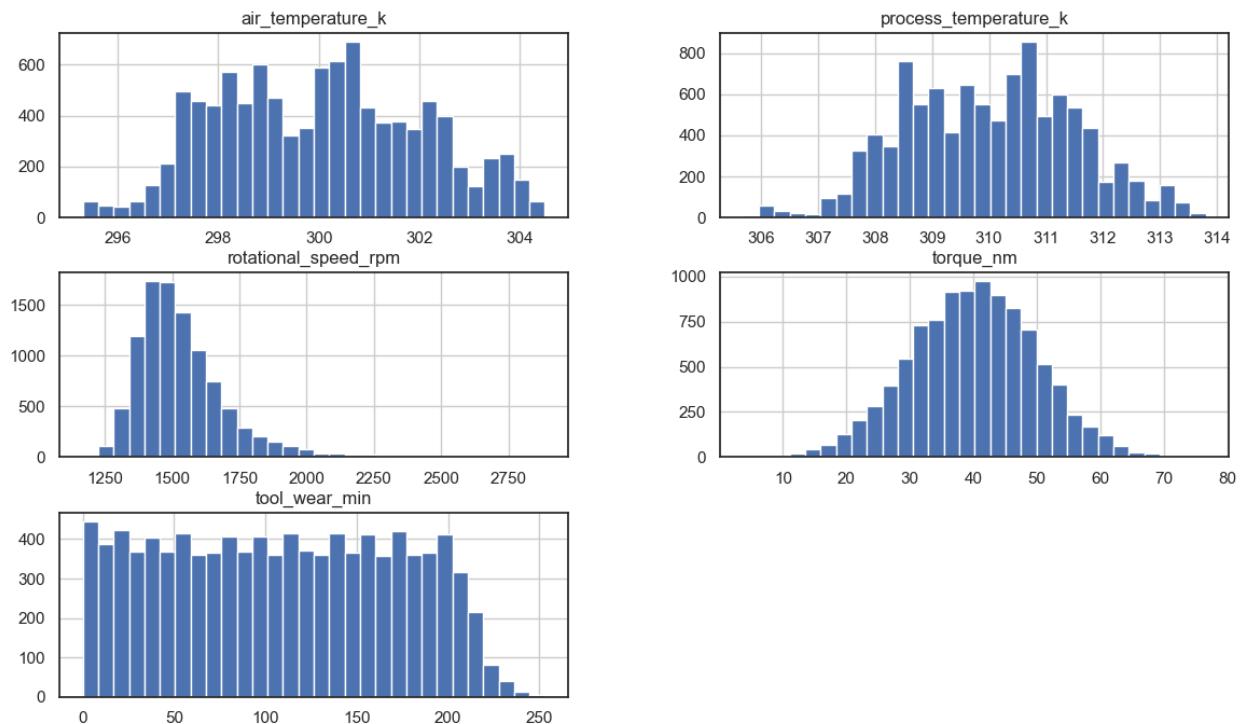
sns.countplot(data=df_eda, x="quality", hue=binary_target)
plt.title("Machine Failure by Product Quality")
plt.show()

# =====
# 14. CORRELATION MATRIX
# =====
corr_matrix = df_eda[num_features].corr().round(3)
display(corr_matrix)

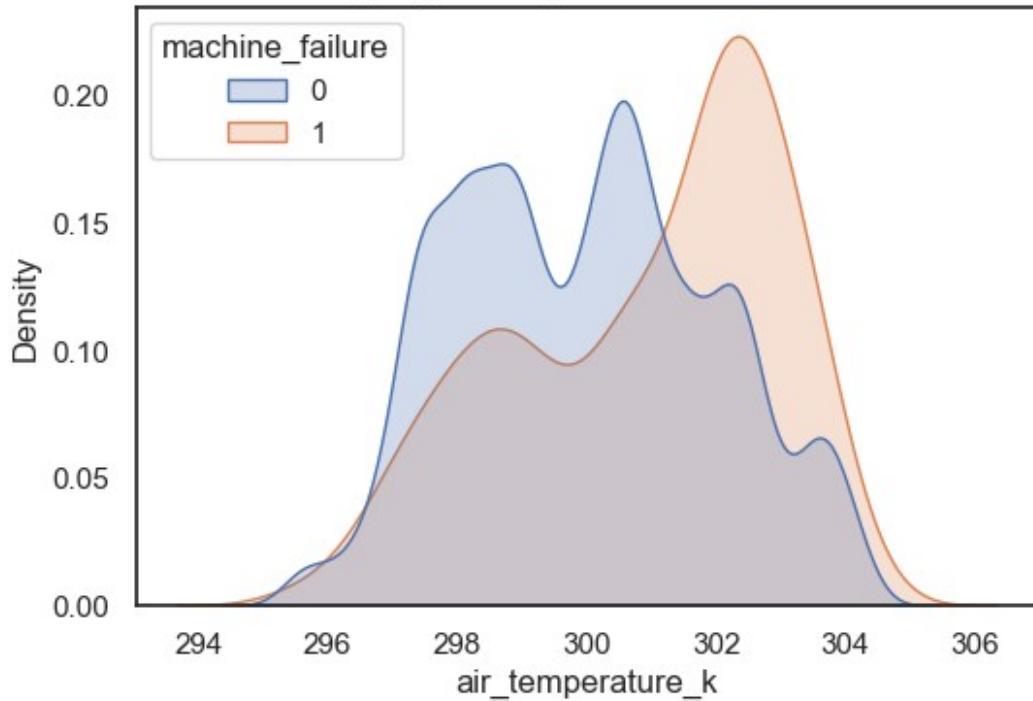
corr_with_failure = (
    df_eda[num_features + [binary_target]]
    .corr()[binary_target]
    .drop(binary_target)
    .sort_values(ascending=False)
    .round(3)
)
display(corr_with_failure)

```

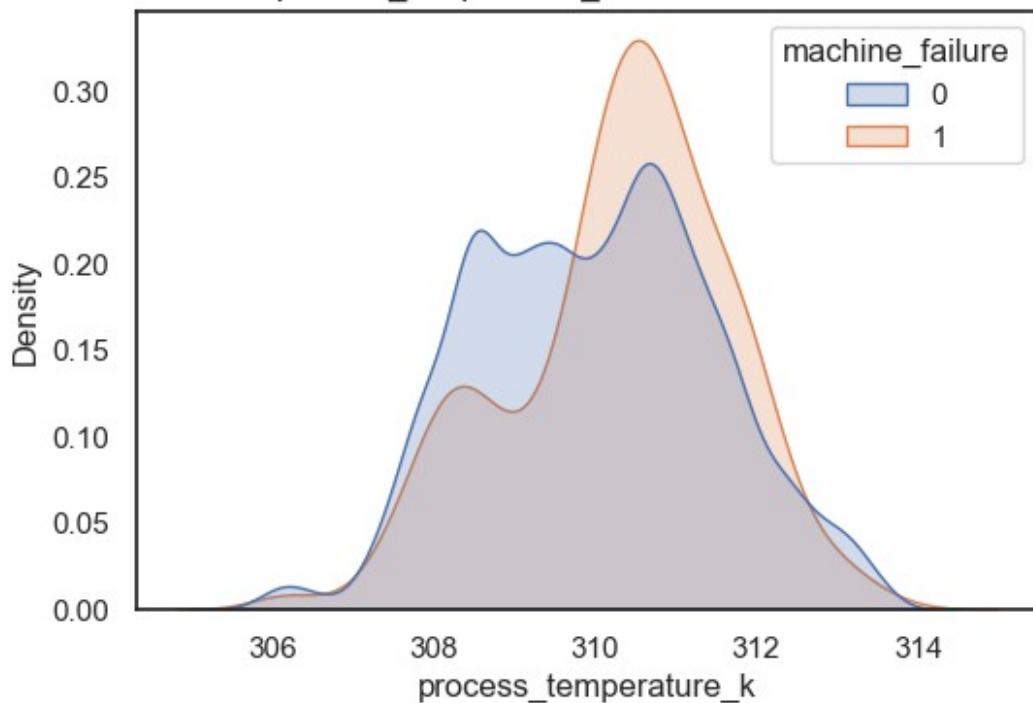
Numerical Feature Distributions



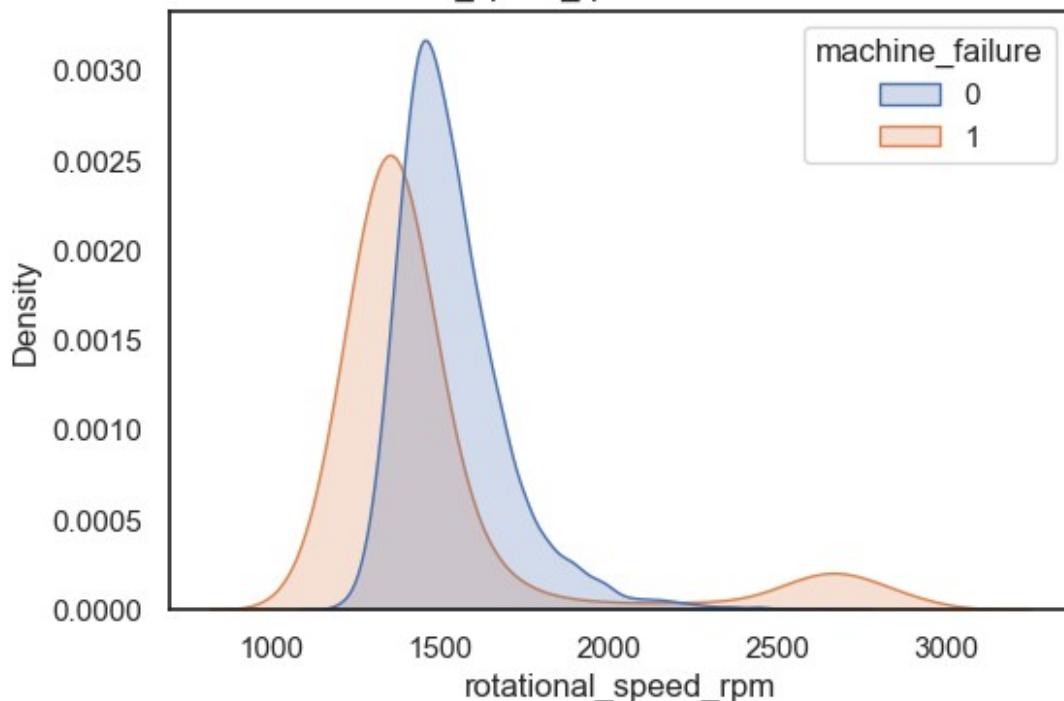
`air_temperature_k` vs Machine Failure



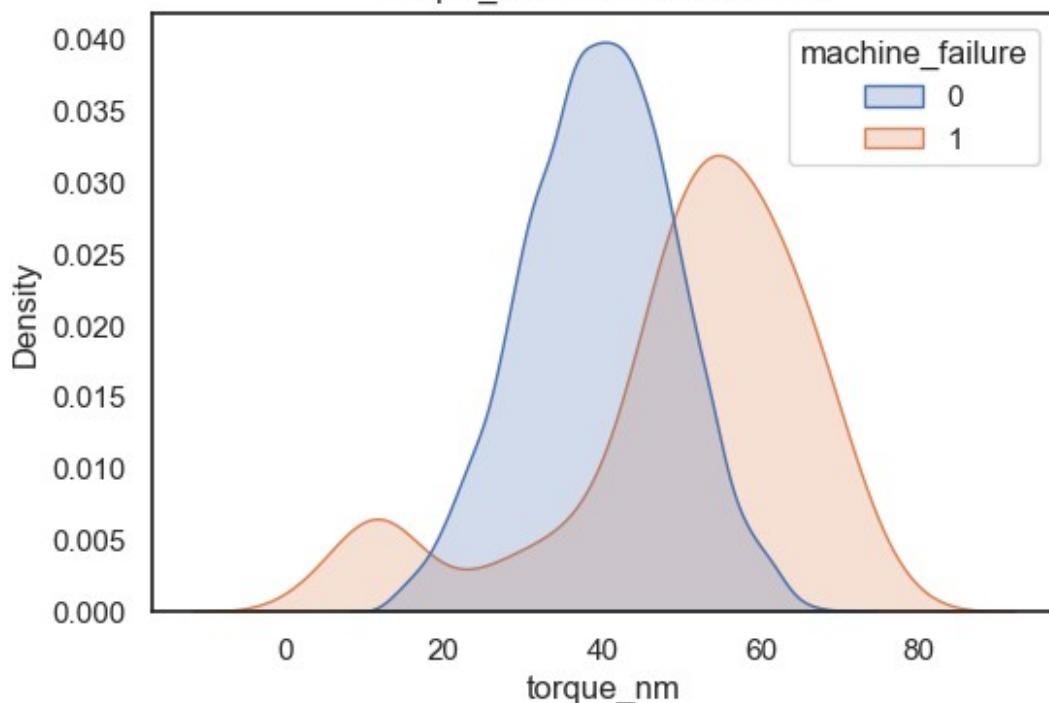
process_temperature_k vs Machine Failure



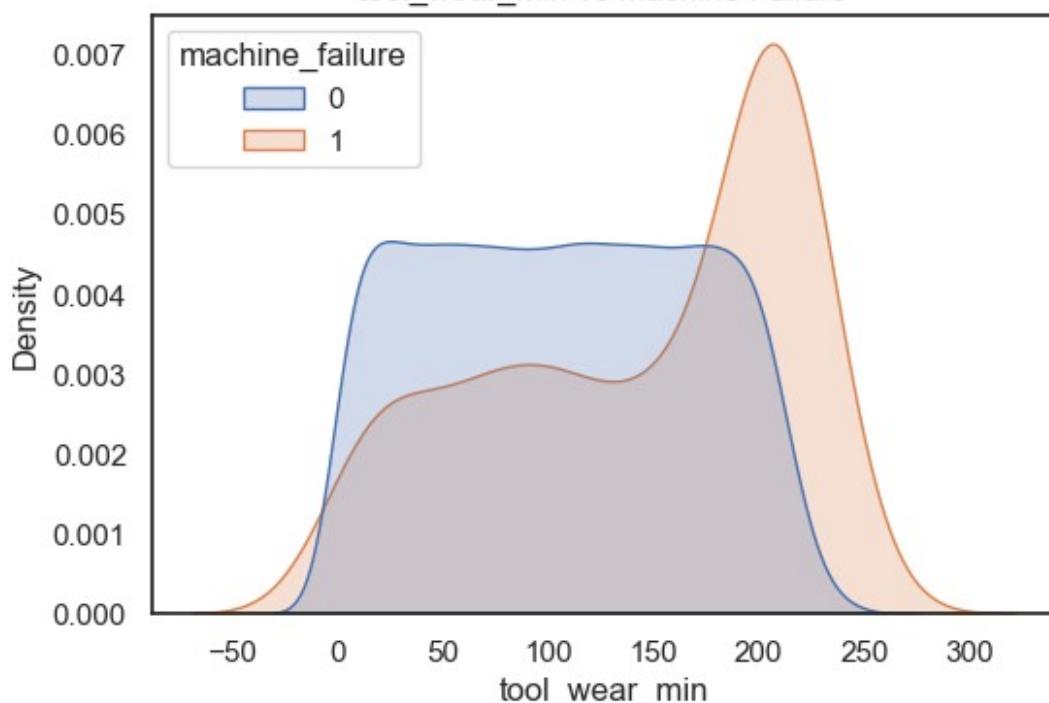
rotational_speed_rpm vs Machine Failure



torque_nm vs Machine Failure



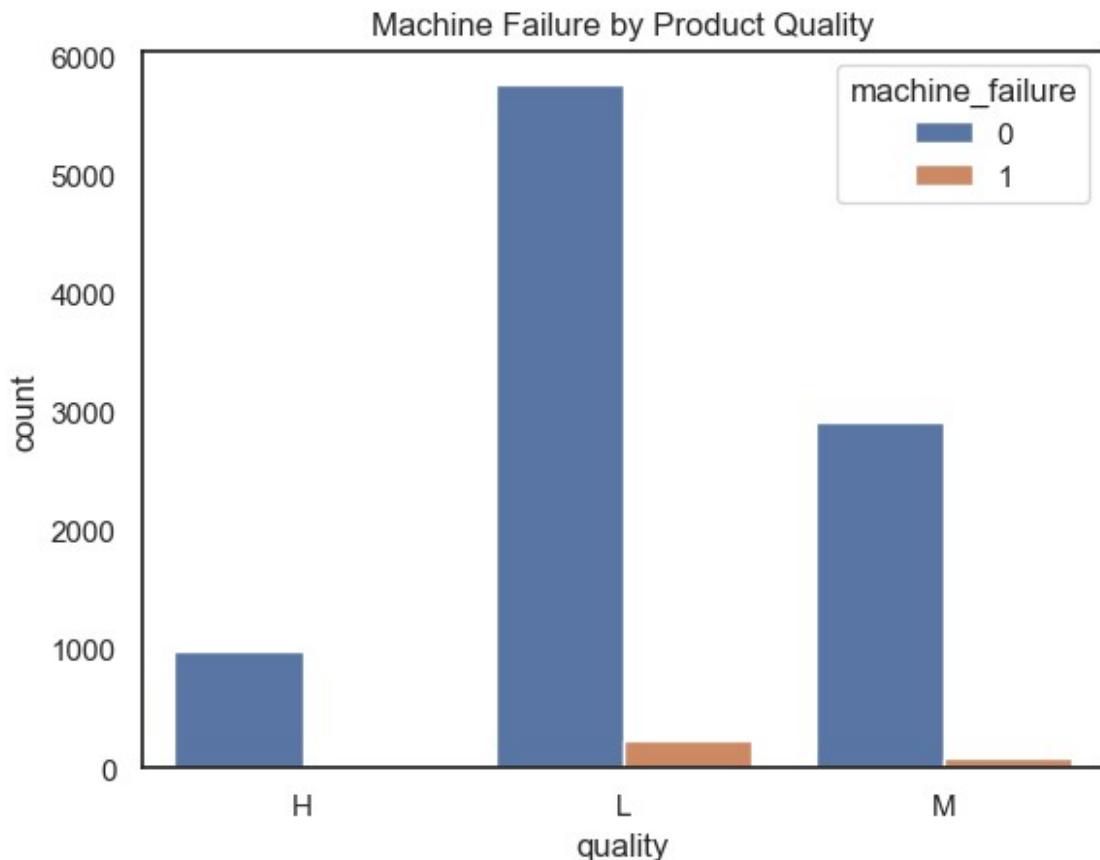
tool_wear_min vs Machine Failure



Failure rate by quality:

machine_failure 0 1

quality			
H	0.979063	0.020937	
L	0.960833	0.039167	
M	0.972306	0.027694	



	air_temperature_k	process_temperature_k	\
air_temperature_k	1.000	0.876	
process_temperature_k	0.876	1.000	
rotational_speed_rpm	0.023	0.019	
torque_nm	-0.014	-0.014	
tool_wear_min	0.014	0.013	
	rotational_speed_rpm	torque_nm	tool_wear_min
air_temperature_k	0.023	-0.014	0.014
process_temperature_k	0.019	-0.014	0.013
rotational_speed_rpm	1.000	-0.875	0.000
torque_nm	-0.875	1.000	-0.003
tool_wear_min	0.000	-0.003	1.000

```

torque_nm          0.191
tool_wear_min      0.105
air_temperature_k   0.083
process_temperature_k 0.036
rotational_speed_rpm -0.044
Name: machine_failure, dtype: float64

```

15-17. Feature design and Scaling

- **Categorical Encoding:** Convert the `quality` feature into dummy variables to make them compatible with Scikit-Learn models.
- **Target Definition:** Separate the data into a binary target (`machine_failure`) and a multi-label target (the five specific failure modes).
- **Feature Scaling:** Apply `StandardScaler` to ensure all numerical features have a mean of 0 and a variance of 1. This is critical for the convergence of Logistic Regression and SVM.

```

# =====
# 15. MODEL-READY DATAFRAMES
# =====
df_model = pd.get_dummies(df_eda, columns=[ "quality" ],
drop_first=True)

X_bin = df_model.drop(columns=[binary_target] + failure_modes)
y_bin = df_model[binary_target]

X_multi = X_bin.copy()
y_multi = df_model[failure_modes]

# =====
# 16. FEATURE SCALING
# =====
scaler = StandardScaler()
X_bin[num_features] = scaler.fit_transform(X_bin[num_features])
X_multi[num_features] = scaler.transform(X_multi[num_features])

# =====
# 17. FINAL DTTRAIN SHAPE CHECK
# =====
print("\nBinary classification shapes:")
print("X_bin:", X_bin.shape)
print("y_bin:", y_bin.shape)

print("\nMulti-label classification shapes:")
print("X_multi:", X_multi.shape)
print("y_multi:", y_multi.shape)

```

```

print("\nFinal feature list:")
print(X_bin.columns.tolist())

Binary classification shapes:
X_bin: (10000, 7)
y_bin: (10000,)

Multi-label classification shapes:
X_multi: (10000, 7)
y_multi: (10000, 5)

Final feature list:
['air_temperature_k', 'process_temperature_k', 'rotational_speed_rpm',
'torque_nm', 'tool_wear_min', 'quality_L', 'quality_M']

```

18-20. Model Preparation and Class Imbalance

- **Stratified Splitting:** We use stratified splits for the binary target to ensure the rare "failure" class (3.4%) is represented equally in training, validation, and test sets.
- **Handling Imbalance:** Instead of oversampling with SMOTE for the binary task, we will utilize model-specific parameters like `class_weight="balanced"` and `scale_pos_weight` to penalize misclassifications of the minority class more heavily.

```

# -----
# 18. TRAIN / VALIDATION / TEST SPLIT
# -----
RANDOM_STATE = 42
# ----- Binary -----
X_train, X_temp, y_train, y_temp = train_test_split(
    X_bin, y_bin,
    test_size=0.3,
    stratify=y_bin,
    random_state=RANDOM_STATE
)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp,
    test_size=0.5,
    stratify=y_temp,
    random_state=RANDOM_STATE
)

# ----- Multi-label -----
X_train_m, X_temp_m, y_train_m, y_temp_m = train_test_split(
    X_multi, y_multi,
    test_size=0.3,
    random_state=RANDOM_STATE
)

```

```

X_val_m, X_test_m, y_val_m, y_test_m = train_test_split(
    X_temp_m, y_temp_m,
    test_size=0.5,
    random_state=RANDOM_STATE
)

# =====
# 19. HELPER FUNCTIONS
# =====

def evaluate_binary(y_true, y_pred, title):
    print(f"\n{title}")
    print("F1-score:", f1_score(y_true, y_pred))
    print(classification_report(y_true, y_pred))

def evaluate_multilabel(y_true, y_pred, title):
    micro = f1_score(y_true, y_pred, average="micro")
    macro = f1_score(y_true, y_pred, average="macro")
    print(f"\n{title}")
    print("Micro F1:", micro)
    print("Macro F1:", macro)

def plot_learning_curve(model, X, y, title):
    from sklearn.model_selection import learning_curve
    train_sizes, train_scores, val_scores = learning_curve(
        model, X, y, cv=5, scoring="f1",
    train_sizes=np.linspace(0.1,1.0,5), n_jobs=-1
    )
    plt.figure(figsize=(6,4))
    plt.plot(train_sizes, train_scores.mean(axis=1), label="Train F1")
    plt.plot(train_sizes, val_scores.mean(axis=1), label="CV F1")
    plt.xlabel("Training size")
    plt.ylabel("F1-score")
    plt.title(title)
    plt.legend()
    plt.show()

# =====
# 20. BALANCE BINARY TRAINING DATA (REMOVED SMOTE)
# =====
# We are dropping SMOTE and using class_weight/scale_pos_weight instead
X_train_bal, y_train_bal = X_train, y_train
print("Class distribution:", y_train_bal.value_counts())

Class distribution: machine_failure
0      6763

```

```
1      237
Name: count, dtype: int64
```

21. Hyperparameter Tuning with Grid Search

We perform a `GridSearchCV` combined with `RepeatedStratifiedKFold` to find the optimal hyperparameters for four different architectures:

1. **Logistic Regression & Linear SVM:** Baseline linear models.
2. **Random Forest:** A non-linear ensemble tuned with deep trees to capture complex interactions.
3. **XGBoost:** A gradient boosting model optimized for high recall on the failure class. Each model is evaluated using the **F1-score**, as it provides a better balance between Precision and Recall than accuracy for imbalanced data.

```
# =====
# 21. GRID SEARCH + REPEATED CROSS-VALIDATION
# =====

cv_strategy = RepeatedStratifiedKFold(n_splits=5, n_repeats=3,
random_state=RANDOM_STATE)

# ----- Logistic Regression -----
print("\n===== GRID SEARCH: LOGISTIC REGRESSION =====")
lr = LogisticRegression(max_iter=5000, class_weight="balanced",
n_jobs=-1)
lr_param_grid = {"C": [0.01, 0.1, 1, 10, 50]}

lr_grid = GridSearchCV(lr, lr_param_grid, scoring="f1",
cv=cv_strategy, n_jobs=-1, verbose=1)
lr_grid.fit(X_train_bal, y_train_bal)

best_lr = lr_grid.best_estimator_
print("Best Logistic Regression params:", lr_grid.best_params_)

evaluate_binary(y_test, best_lr.predict(X_test), "Logistic Regression
(Binary)")
plot_learning_curve(best_lr, X_train_bal, y_train_bal, "Logistic
Regression Learning Curve")

# ----- Linear SVM -----
print("\n===== GRID SEARCH: LINEAR SVM =====")
svm = LinearSVC(max_iter=5000, class_weight="balanced")
svm_param_grid = {"C": [0.01, 0.1, 1, 10, 50]}

svm_grid = GridSearchCV(svm, svm_param_grid, scoring="f1",
cv=cv_strategy, n_jobs=-1, verbose=1)
svm_grid.fit(X_train_bal, y_train_bal)
```

```

best_svm = svm_grid.best_estimator_
print("Best Linear SVM params:", svm_grid.best_params_)

evaluate_binary(y_test, best_svm.predict(X_test), "Linear SVM
(Binary)")
plot_learning_curve(best_svm, X_train_bal, y_train_bal, "Linear SVM
Learning Curve")

# ----- Random Forest: Pushing Depth -----
print("\n===== GRID SEARCH: RANDOM FOREST =====")
rf = RandomForestClassifier(random_state=RANDOM_STATE,
class_weight="balanced", n_jobs=-1)
rf_param_grid = {
    "n_estimators": [300, 500],
    "max_depth": [15, 20, 25],      # Pushing depth further
    "min_samples_split": [10, 20],  # More aggressive splitting
    "min_samples_leaf": [5, 8]     # Smaller leaves to capture subtle
signals
}

rf_grid = GridSearchCV(rf, rf_param_grid, scoring="f1",
cv=cv_strategy, n_jobs=-1, verbose=1)
rf_grid.fit(X_train_bal, y_train_bal)

best_rf = rf_grid.best_estimator_
print("Best Random Forest params:", rf_grid.best_params_)

evaluate_binary(y_test, best_rf.predict(X_test), "Random Forest
(Binary)")
plot_learning_curve(best_rf, X_train_bal, y_train_bal, "Random Forest
Learning Curve")

# ----- XGBoost: Minimizing Regularization -----
print("\n===== GRID SEARCH: XGBOOST =====")
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    eval_metric="logloss",
    random_state=RANDOM_STATE,
    n_jobs=-1
)

xgb_param_grid = {
    "n_estimators": [500],          # More trees for finer detail
    "max_depth": [7, 8],           # Deepest trees yet
    "learning_rate": [0.03, 0.05], # Slightly slower learning for the
extra depth
    "subsample": [0.9],            # See more data
    "colsample_bytree": [0.9],
}

```

```

        "gamma": [0.1, 0.2],                      # Very low gamma (almost no split
penalty)
        "reg_lambda": [0.1, 1],                      # Very low L2 regularization
        "scale_pos_weight": [28]
    }

xgb_grid = GridSearchCV(xgb_model, xgb_param_grid, scoring="f1",
cv=cv_strategy, n_jobs=-1, verbose=1)
xgb_grid.fit(X_train_bal, y_train_bal)

best_xgb = xgb_grid.best_estimator_
print("Best XGBoost params:", xgb_grid.best_params_)

evaluate_binary(y_test, best_xgb.predict(X_test), "XGBoost (Binary)")
plot_learning_curve(best_xgb, X_train_bal, y_train_bal, "XGBoost
Learning Curve")

# =====
# 22. MULTI-LABEL CLASSIFICATION WITH XGBOOST
# =====

multi_models = {}
y_pred_multi = pd.DataFrame(index=X_test_m.index)

for mode in failure_modes:
    smote = SMOTE(random_state=RANDOM_STATE)
    X_train_m_bal, y_train_m_bal = smote.fit_resample(X_train_m,
y_train_m[mode])

    model = xgb.XGBClassifier(
        objective="binary:logistic",
        eval_metric="logloss",
        use_label_encoder=False,
        n_estimators=200,
        max_depth=4,
        learning_rate=0.05,
        scale_pos_weight=int((y_train_m[mode]==0).sum() /
(y_train_m[mode]==1).sum()),
        random_state=RANDOM_STATE,
        n_jobs=-1
    )
    model.fit(X_train_m_bal, y_train_m_bal)
    multi_models[mode] = model
    y_pred_multi[mode] = model.predict(X_test_m)

evaluate_multilabel(y_test_m, y_pred_multi, "XGBoost Multi-label
Classification")

```

```

# =====
# 23. OPTIONAL VOTING ENSEMBLE (BINARY)
# =====

ensemble = VotingClassifier(
    estimators=[
        ('lr', best_lr),
        ('rf', best_rf),
        ('xgb', best_xgb)
    ],
    voting='soft',
    n_jobs=-1
)

ensemble.fit(X_train_bal, y_train_bal)
evaluate_binary(y_test, ensemble.predict(X_test), "Voting Ensemble (Binary)")

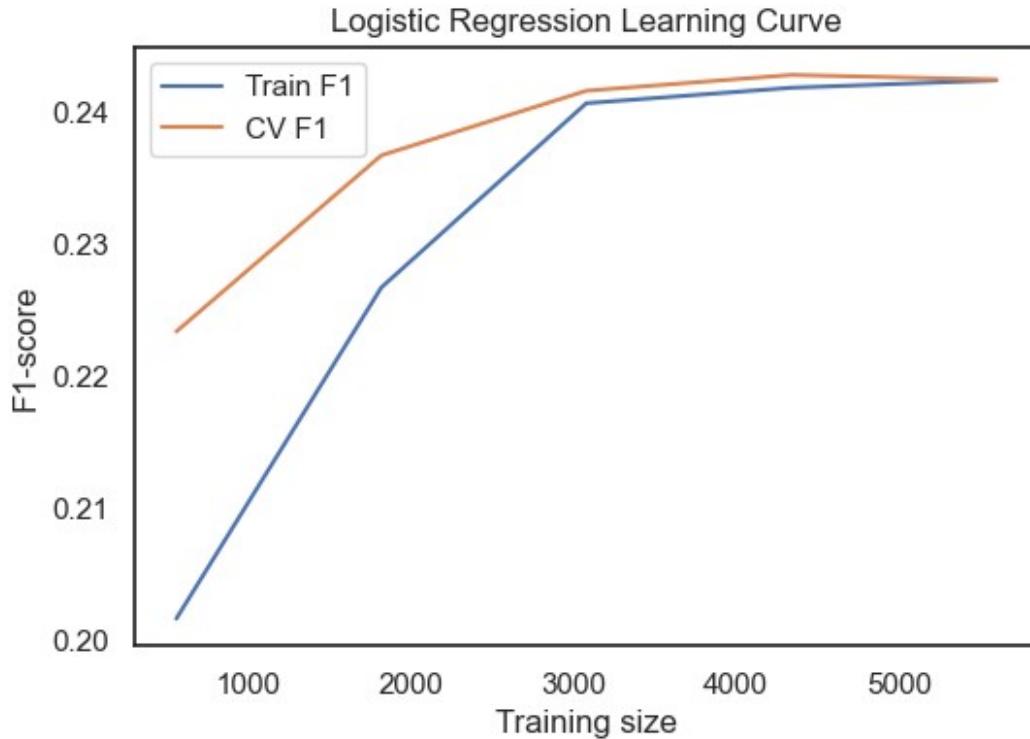
```

===== GRID SEARCH: LOGISTIC REGRESSION =====
Fitting 15 folds for each of 5 candidates, totalling 75 fits
Best Logistic Regression params: {'C': 0.1}

Logistic Regression (Binary)

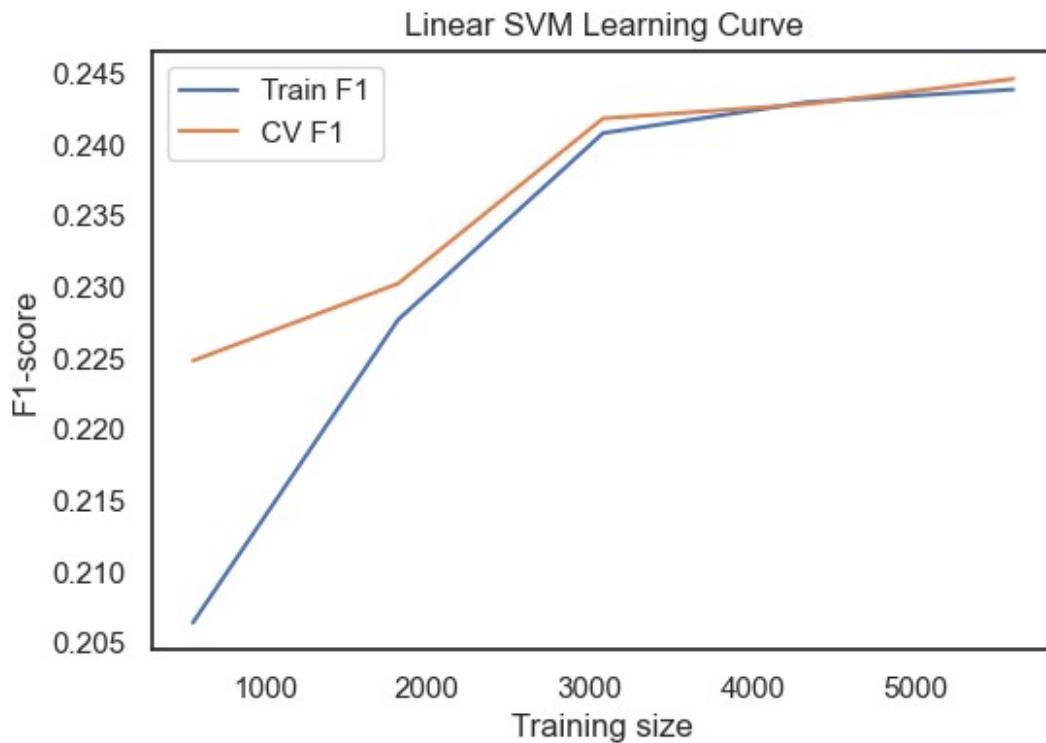
F1-score: 0.25

	precision	recall	f1-score	support
0	0.99	0.82	0.90	1449
1	0.15	0.86	0.25	51
accuracy			0.82	1500
macro avg	0.57	0.84	0.58	1500
weighted avg	0.97	0.82	0.88	1500



```
===== GRID SEARCH: LINEAR SVM =====
Fitting 15 folds for each of 5 candidates, totalling 75 fits
Best Linear SVM params: {'C': 1}
```

```
Linear SVM (Binary)
F1-score: 0.25142857142857145
      precision    recall  f1-score   support
          0       0.99     0.82    0.90    1449
          1       0.15     0.86    0.25      51
   accuracy                           0.83    1500
  macro avg       0.57     0.84    0.58    1500
weighted avg       0.97     0.83    0.88    1500
```

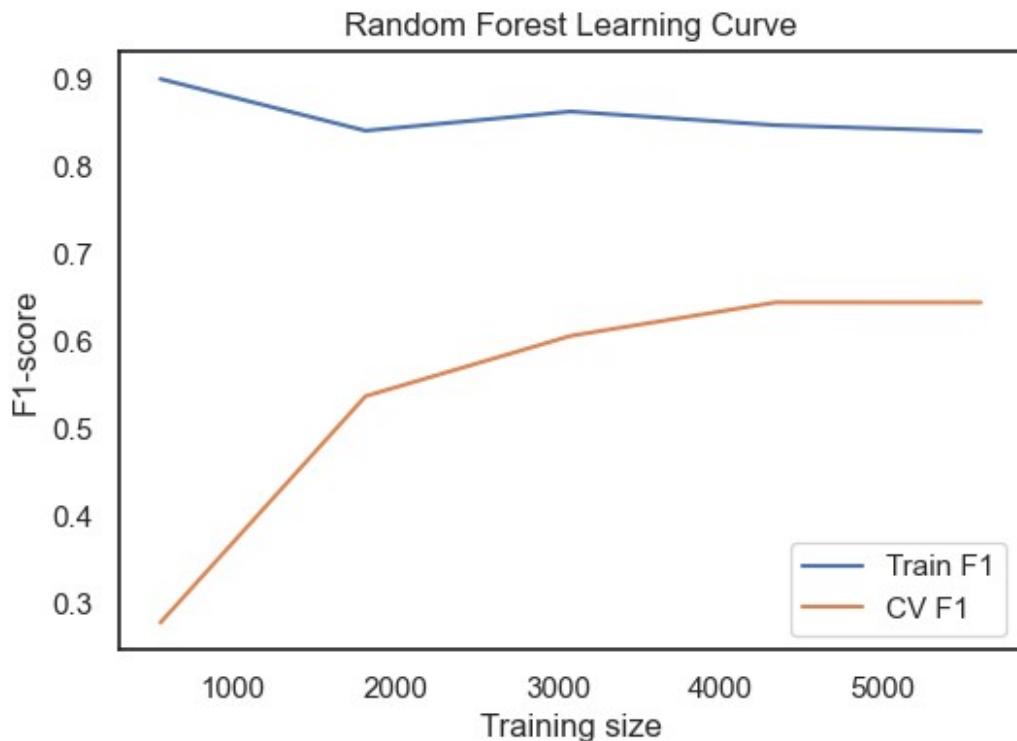


```
===== GRID SEARCH: RANDOM FOREST =====
Fitting 15 folds for each of 24 candidates, totalling 360 fits
Best Random Forest params: {'max_depth': 20, 'min_samples_leaf': 5,
'min_samples_split': 10, 'n_estimators': 500}
```

Random Forest (Binary)

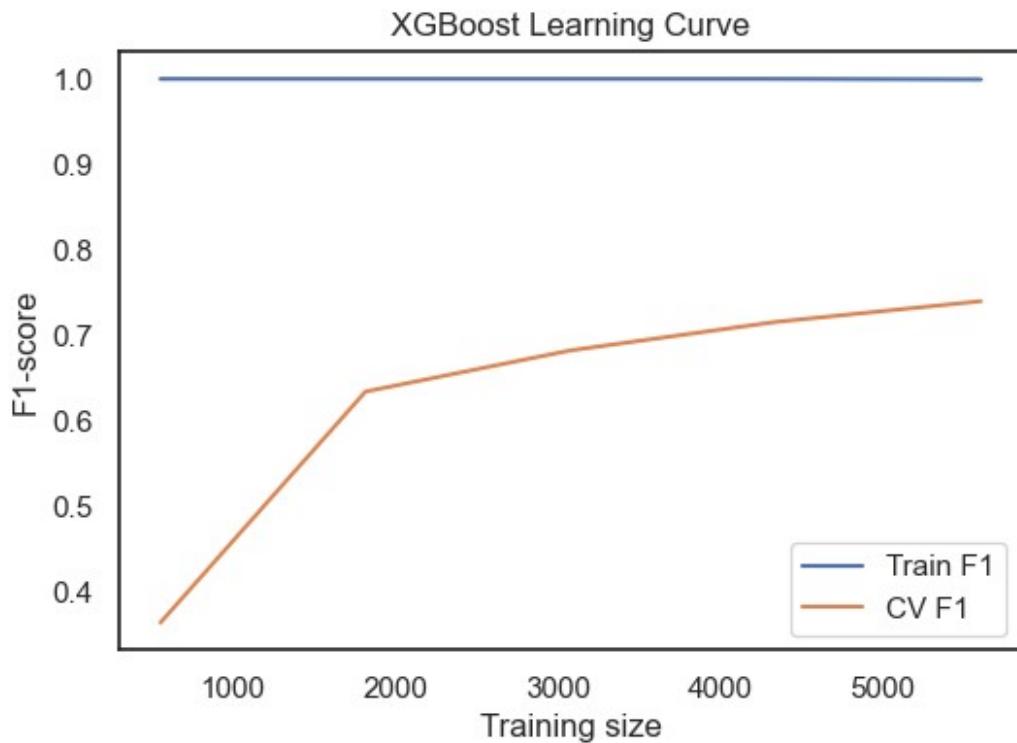
F1-score: 0.6446280991735537

	precision	recall	f1-score	support
0	0.99	0.98	0.99	1449
1	0.56	0.76	0.64	51
accuracy			0.97	1500
macro avg	0.77	0.87	0.81	1500
weighted avg	0.98	0.97	0.97	1500



```
===== GRID SEARCH: XGBOOST =====
Fitting 15 folds for each of 16 candidates, totalling 240 fits
Best XGBoost params: {'colsample_bytree': 0.9, 'gamma': 0.1,
'learning_rate': 0.03, 'max_depth': 8, 'n_estimators': 500,
'reg_lambda': 0.1, 'scale_pos_weight': 28, 'subsample': 0.9}

XGBoost (Binary)
F1-score: 0.7551020408163265
      precision    recall  f1-score   support
          0       0.99     0.99     0.99     1449
          1       0.79     0.73     0.76      51
   accuracy                           0.98     1500
  macro avg       0.89     0.86     0.87     1500
weighted avg       0.98     0.98     0.98     1500
```



XGBoost Multi-label Classification

Micro F1: 0.1700404858299595

Macro F1: 0.5190681194798756

Voting Ensemble (Binary)

F1-score: 0.7130434782608696

	precision	recall	f1-score	support
0	0.99	0.98	0.99	1449
1	0.64	0.80	0.71	51
accuracy			0.98	1500
macro avg	0.82	0.89	0.85	1500
weighted avg	0.98	0.98	0.98	1500