

Salem Ahmed Buhluli - 262000880

Electric Power Substations - Machine Learning Project

Dataset: Substations.csv (Kaggle / HIFLD Electric Power Substations)

Assignment steps:

1. Download a dataset from Kaggle
2. Handle missing values appropriately
3. Apply machine learning models with a **grid search optimizer** and evaluate performance
4. Use **two feature selection methods** to identify the most important features
5. Apply machine learning models to the selected features and evaluate
6. Save the code file as full name and upload as PDF

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, confusion_matrix,
                             classification_report, RocCurveDisplay)

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)
sns.set_style('whitegrid')

# --- Make wide tables readable in the exported PDF ---
from IPython.display import display, HTML
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 120)
display(HTML(
    '<style>'
    'table.dataframe { font-size: 9px !important; width: 100% !important;'
    ' table-layout: fixed; word-wrap: break-word; }'
    'table.dataframe th, table.dataframe td { padding: 2px 3px !important;'
```

```
' white-space: normal !important; overflow-wrap: anywhere; }'  
'</style>'))
```

Step 1: Load the dataset

```
In [2]: df = pd.read_csv('Desktop/Substations.csv', low_memory=False)  
print('Shape:', df.shape)  
print('Rows:', df.shape[0], ' | Columns:', df.shape[1])
```

Shape: (79687, 26)
Rows: 79687 | Columns: 26

```
In [3]: # The dataset has 26 columns, which is too wide to print across a page,  
# so we show the first rows TRANSPOSED (columns become rows).  
df.head(3).T
```

```
Out[3]:
```

	0	1	2
X	-10226336.5648	-10053464.9584	-10144276.0661
Y	5743316.2456	5706699.7421	5656784.2806
OBJECTID	1	2	3
ID	107655	107656	107657
NAME	UNKNOWN107655	UNKNOWN107656	HOLCOMBE
CITY	SHELL LAKE	PRENTICE	HOLCOMBE
STATE	WI	WI	WI
ZIP	54871	54556	54745
TYPE	SUBSTATION	SUBSTATION	SUBSTATION
STATUS	IN SERVICE	IN SERVICE	IN SERVICE
COUNTY	WASHBURN	PRICE	CHIPPEWA
COUNTYFIPS	55129	55099	55017
COUNTRY	USA	USA	USA
LATITUDE	45.768423	45.538502	45.223553
LONGITUDE	-91.864744	-90.311812	-91.127582
NAICS_CODE	221121	221121	221121
NAICS_DESC	ELECTRIC BULK POWER TRANSMISSION AND CONTROL	ELECTRIC BULK POWER TRANSMISSION AND CONTROL	ELECTRIC BULK POWER TRANSMISSION AND CONTROL
SOURCE	http://psc.wi.gov/utilityinfo/maps/ electricmap...	http://psc.wi.gov/utilityinfo/maps/ electricmap...	http://psc.wi.gov/utilityinfo/maps/ electricmap...
SOURCEDATE	2014/04/07 00:00:00+00	2014/04/07 00:00:00+00	2014/04/07 00:00:00+00
VAL_METHOD	IMAGERY	IMAGERY	IMAGERY
VAL_DATE	2015/08/28 00:00:00+00	2015/08/28 00:00:00+00	2021/06/01 00:00:00+00
LINES	5	3	4
MAX_VOLT	161.0	115.0	115.0
MIN_VOLT	69.0	69.0	115.0
MAX_INFER	N	N	Y
MIN_INFER	Y	Y	Y

```
In [4]: df.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 79687 entries, 0 to 79686
Data columns (total 26 columns):
#   Column      Non-Null Count  Dtype
---  -
0   X            79687 non-null  float64
1   Y            79687 non-null  float64
2   OBJECTID     79687 non-null  int64
3   ID           79687 non-null  int64
4   NAME         79687 non-null  object
5   CITY         79687 non-null  object
6   STATE        79687 non-null  object
7   ZIP          79687 non-null  object
8   TYPE         79687 non-null  object
9   STATUS       79687 non-null  object
10  COUNTY       79687 non-null  object
11  COUNTYFIPS   79687 non-null  object
12  COUNTRY      79687 non-null  object
13  LATITUDE     79687 non-null  float64
14  LONGITUDE    79687 non-null  float64
15  NAICS_CODE   79687 non-null  int64
16  NAICS_DESC   79687 non-null  object
17  SOURCE       79687 non-null  object
18  SOURCEDATE   79687 non-null  object
19  VAL_METHOD   79687 non-null  object
20  VAL_DATE     79687 non-null  object
21  LINES        79687 non-null  int64
22  MAX_VOLT     79687 non-null  float64
23  MIN_VOLT     79687 non-null  float64
24  MAX_INFER    79687 non-null  object
25  MIN_INFER    79687 non-null  object
dtypes: float64(6), int64(4), object(16)
memory usage: 15.8+ MB

```

```

In [5]: # describe() is also transposed so every statistic stays inside the page
df.describe().T

```

```

Out[5]:

```

	count	mean	std	min	25%	50%	75%	max
X	79687.0	-1.033163e+07	1.666416e+06	-1.841563e+07	-1.106731e+07	-9.953265e+06	-9.162144e+06	1.622301e+07
Y	79687.0	4.647277e+06	7.948806e+05	1.493168e+06	4.064301e+06	4.639909e+06	5.192541e+06	1.117329e+07
OBJECTID	79687.0	3.984400e+04	2.300380e+04	1.000000e+00	1.992250e+04	3.984400e+04	5.976550e+04	7.968700e+04
ID	79687.0	1.744173e+05	6.031313e+04	1.076550e+05	1.338425e+05	1.553530e+05	2.013940e+05	3.116560e+05
LATITUDE	79687.0	3.826449e+01	5.545430e+00	1.329248e+01	3.426397e+01	3.842690e+01	4.221035e+01	7.031883e+01
LONGITUDE	79687.0	-9.281063e+01	1.496967e+01	-1.654304e+02	-9.941938e+01	-8.941170e+01	-8.230494e+01	1.457337e+02
NAICS_CODE	79687.0	2.211210e+05	0.000000e+00	2.211210e+05	2.211210e+05	2.211210e+05	2.211210e+05	2.211210e+05
LINES	79687.0	2.298543e+00	1.896546e+00	0.000000e+00	1.000000e+00	2.000000e+00	3.000000e+00	3.000000e+01
MAX_VOLT	79687.0	-2.455933e+05	4.305485e+05	-9.999990e+05	3.450000e+01	6.900000e+01	1.380000e+02	1.000000e+03
MIN_VOLT	79687.0	-2.683844e+05	4.432095e+05	-9.999990e+05	-9.999990e+05	6.900000e+01	1.150000e+02	7.650000e+02

Step 2: Handle Missing Values

This dataset does **not** use blank cells. Instead it encodes missing data with **sentinel values**:

- -999999 in the numeric voltage columns (MAX_VOLT , MIN_VOLT)
- the string 'NOT AVAILABLE' in categorical columns

So a plain `isnull()` reports zero missing values, which is misleading. We must convert these sentinels to real `NaN` first.

```
In [6]: # A plain missing-value check is misleading here
print('Missing values reported by isnull():', df.isnull().sum().sum())
```

Missing values reported by isnull(): 0

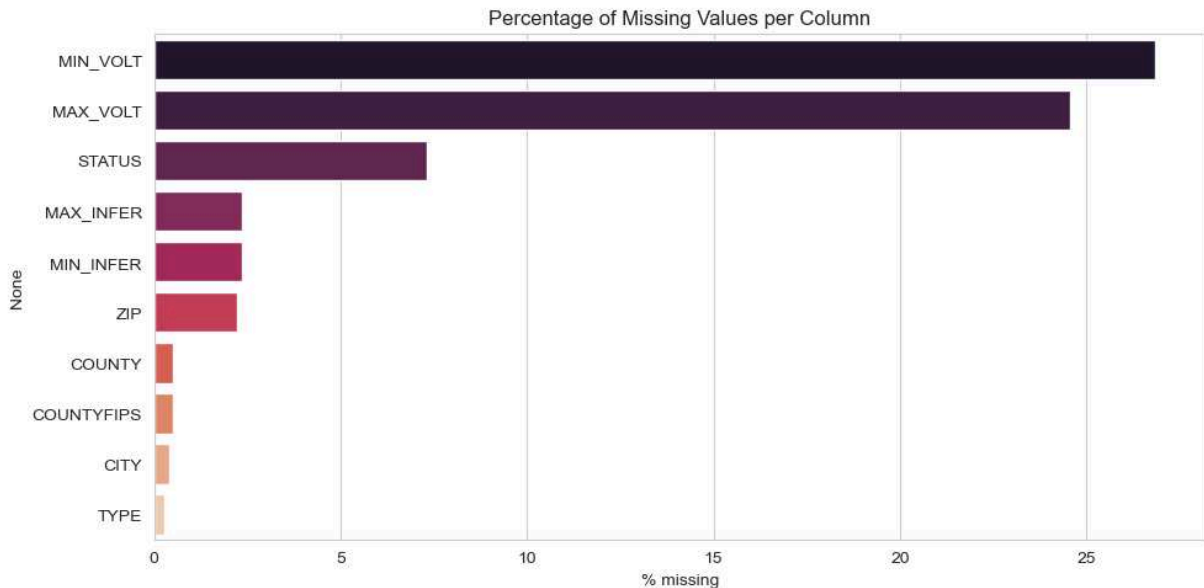
```
In [7]: # Convert the hidden sentinel values into real NaN
df = df.replace(-999999, np.nan)
df = df.replace(['NOT AVAILABLE', 'NOT-AVAILABLE', 'UNKNOWN', 'NULL', ''], np.nan)

missing = df.isnull().sum()
missing_pct = (df.isnull().mean() * 100).round(2)
report = pd.DataFrame({'missing_count': missing, 'missing_%': missing_pct})
report[report['missing_count'] > 0].sort_values('missing_%', ascending=False)
```

```
Out[7]:
```

	missing_count	missing_%
MIN_VOLT	21393	26.85
MAX_VOLT	19578	24.57
STATUS	5826	7.31
MAX_INFER	1869	2.35
MIN_INFER	1870	2.35
ZIP	1782	2.24
COUNTY	415	0.52
COUNTYFIPS	415	0.52
CITY	321	0.40
TYPE	224	0.28

```
In [8]: # Visualise the missing values
plt.figure(figsize=(10, 5))
miss = missing_pct[missing_pct > 0].sort_values(ascending=False)
sns.barplot(x=miss.values, y=miss.index, palette='rocket')
plt.title('Percentage of Missing Values per Column')
plt.xlabel('% missing')
plt.tight_layout()
plt.show()
```



```
In [9]: # X and Y are the same locations in a projected (Web Mercator) coordinate system,
# so they are perfectly redundant with LONGITUDE / LATITUDE.
print('corr(X, LONGITUDE):', round(df['X'].corr(df['LONGITUDE']), 4))
print('corr(Y, LATITUDE) :', round(df['Y'].corr(df['LATITUDE']), 4))
```

```
corr(X, LONGITUDE): 1.0
corr(Y, LATITUDE) : 0.9974
```

```
In [10]: # Drop identifier / constant / redundant / very-high-cardinality columns
drop_cols = ['OBJECTID', 'ID', 'NAME',           # unique identifiers
             'CITY', 'ZIP', 'COUNTY', 'COUNTYFIPS', # very high cardinality
             'NAICS_CODE', 'NAICS_DESC',         # constant for every row
             'SOURCE', 'SOURCEDATE', 'VAL_DATE', # metadata / free-text dates
             'X', 'Y']                          # projected duplicates of LON/L
df = df.drop(columns=[col for col in drop_cols if col in df.columns])
print('Remaining columns:', list(df.columns))
```

```
Remaining columns: ['STATE', 'TYPE', 'STATUS', 'COUNTRY', 'LATITUDE', 'LONGITUDE',
'VAL_METHOD', 'LINES', 'MAX_VOLT', 'MIN_VOLT', 'MAX_INFER', 'MIN_INFER']
```

```
In [11]: # Drop columns with more than 30% missing values
threshold = 30
missing_percentage = df.isnull().mean() * 100
columns_to_drop = missing_percentage[missing_percentage > threshold].index
df = df.drop(columns=columns_to_drop)
print(f'Columns dropped (>{threshold}% missing): {list(columns_to_drop)}')
print(f'Shape now: {df.shape}')
```

```
Columns dropped (>30% missing): []
Shape now: (79687, 12)
```

Define the target variable

We predict the **facility TYPE**: SUBSTATION vs TAP — the two dominant classes.

```
In [12]: print(df['TYPE'].value_counts(dropna=False))
```

```
# Keep only the two main classes for a clean binary classification problem
df = df[df['TYPE'].isin(['SUBSTATION', 'TAP'])].copy()
df['TYPE'] = (df['TYPE'] == 'SUBSTATION').astype(int) # 1 = SUBSTATION, 0 = TAP
print('\nAfter filtering -> shape:', df.shape)
print(df['TYPE'].value_counts())
```

```
TYPE
SUBSTATION    57027
TAP            21175
DEAD END       754
RISER          507
NaN            224
Name: count, dtype: int64
```

After filtering -> shape: (78202, 12)

```
TYPE
1    57027
0    21175
Name: count, dtype: int64
```

```
In [13]: # Impute the remaining missing values:
#   numeric    -> median (robust to outliers)
#   categorical -> mode (most frequent value)
num_cols_all = df.select_dtypes(include=[np.number]).columns
cat_cols_all = df.select_dtypes(include=['object']).columns

for col in num_cols_all:
    if df[col].isnull().any():
        df[col] = df[col].fillna(df[col].median())

for col in cat_cols_all:
    if df[col].isnull().any():
        df[col] = df[col].fillna(df[col].mode()[0])

print('Total missing values after imputation:', df.isnull().sum().sum())
```

Total missing values after imputation: 0

```
In [14]: # Encode the remaining categorical columns
label_encoder = LabelEncoder()
for col in df.select_dtypes(include=['object']).columns:
    df[col] = label_encoder.fit_transform(df[col].astype(str))

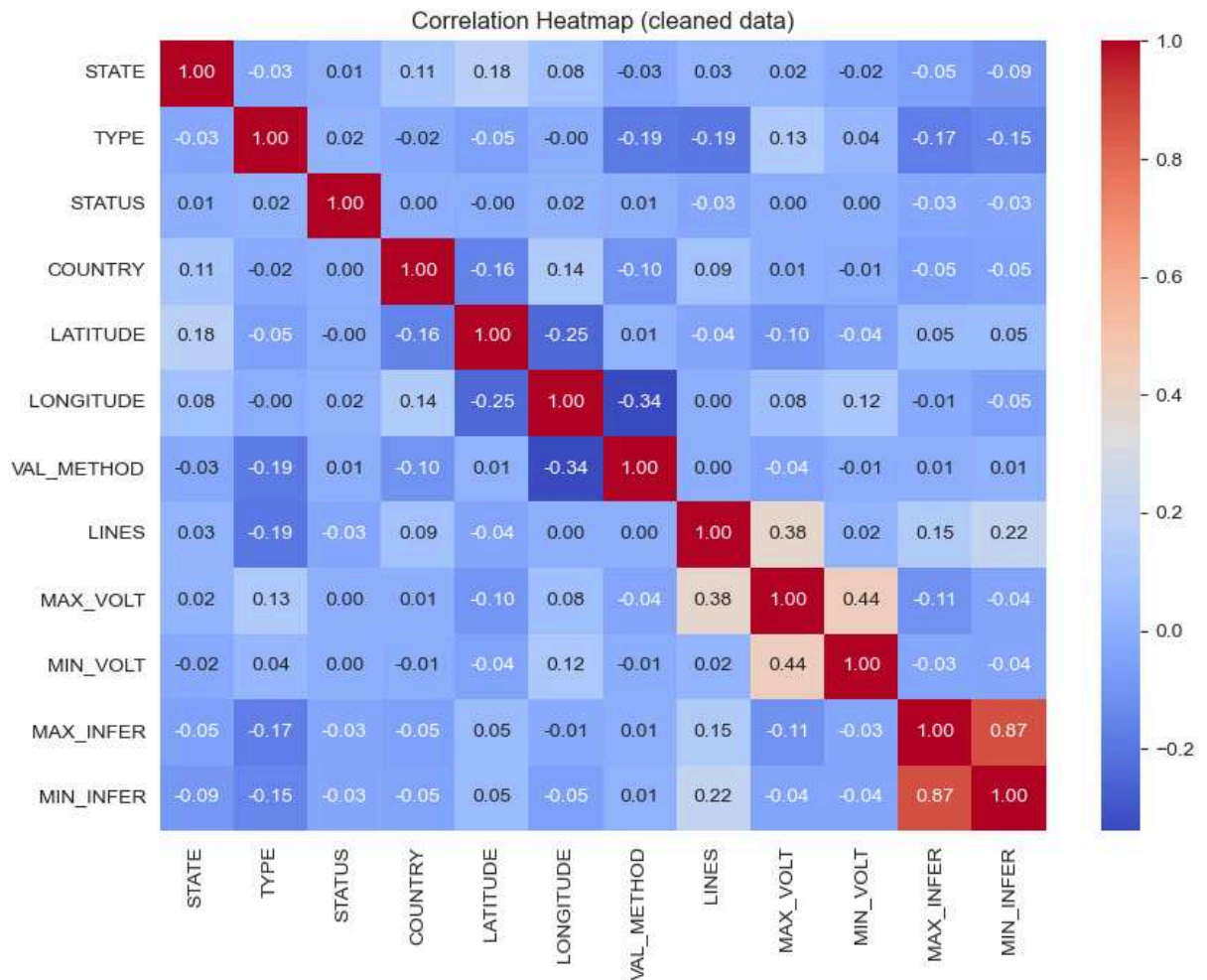
df.head()
```

```
Out[14]:
```

	STATE	TYPE	STATUS	COUNTR Y	LATITUD E	LONGIT UDE	VAL_ME THOD	LINES	MAX_VO LT	MIN_VO LT	MAX_IN FER	MIN_INF ER
0	56	1	0	2	45.76842 3	-91.8647 44	0	5	161.0	69.0	0	1
1	56	1	0	2	45.53850 2	-90.3118 12	0	3	115.0	69.0	0	1
2	56	1	0	2	45.22355 3	-91.1275 82	0	4	115.0	115.0	1	1
3	56	1	0	2	43.71099 3	-87.8383 96	0	3	138.0	69.0	1	1
4	25	1	0	2	45.51378 1	-87.8009 21	0	3	138.0	69.0	0	1

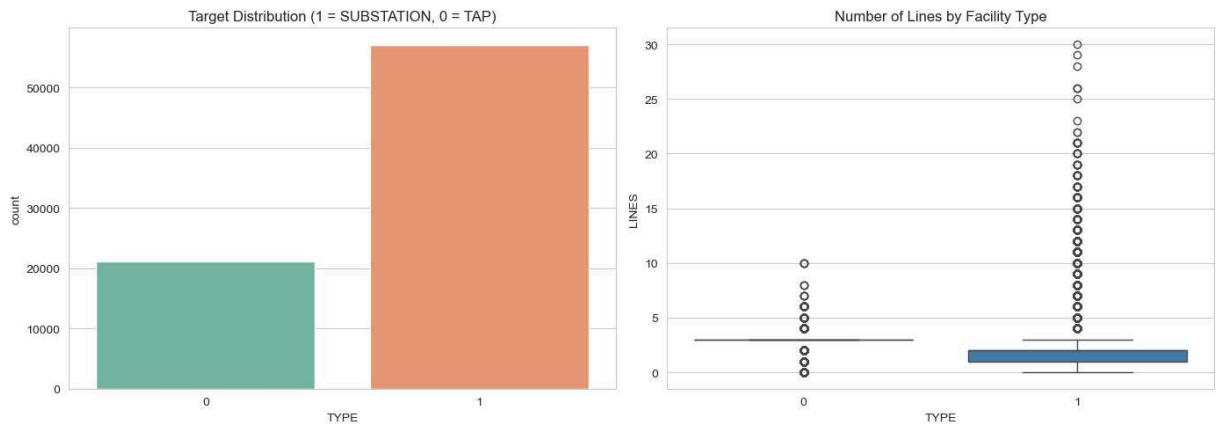
Exploratory visualisation of the cleaned data

```
In [15]: plt.figure(figsize=(9, 7))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap (cleaned data)')
plt.tight_layout()
plt.show()
```



```
In [16]: fig, axes = plt.subplots(1, 2, figsize=(14, 5))
sns.countplot(x='TYPE', data=df, palette='Set2', ax=axes[0])
axes[0].set_title('Target Distribution (1 = SUBSTATION, 0 = TAP)')
axes[0].set_xlabel('TYPE')

sns.boxplot(x='TYPE', y='LINES', data=df, palette='Set1', ax=axes[1])
axes[1].set_title('Number of Lines by Facility Type')
plt.tight_layout()
plt.show()
```



Step 3: Machine Learning Models with Grid Search

We split the data, scale the features, then tune four classifiers with `GridSearchCV` (5-fold cross-validation).

```
In [17]: X = df.drop(columns=['TYPE'])
y = df['TYPE']
print('Features:', list(X.columns))
print('X shape:', X.shape, '| y shape:', y.shape)
```

```
Features: ['STATE', 'STATUS', 'COUNTRY', 'LATITUDE', 'LONGITUDE', 'VAL_METHOD', 'LINES', 'MAX_VOLT', 'MIN_VOLT', 'MAX_INFER', 'MIN_INFER']
X shape: (78202, 11) | y shape: (78202,)
```

```
In [18]: # A stratified subsample keeps the grid search runtime reasonable
SAMPLE_SIZE = 25000
if len(X) > SAMPLE_SIZE:
    X_s, _, y_s, _ = train_test_split(X, y, train_size=SAMPLE_SIZE,
                                      stratify=y, random_state=RANDOM_STATE)
else:
    X_s, y_s = X, y

X_train, X_test, y_train, y_test = train_test_split(
    X_s, y_s, test_size=0.2, stratify=y_s, random_state=RANDOM_STATE)

scaler = StandardScaler()
X_train_sc = scaler.fit_transform(X_train)
X_test_sc = scaler.transform(X_test)

print('Train:', X_train.shape, '| Test:', X_test.shape)
```

```
Train: (20000, 11) | Test: (5000, 11)
```

```
In [19]: # Models and their hyper-parameter grids
models_and_grids = {
    'Logistic Regression': (
        LogisticRegression(max_iter=1000, random_state=RANDOM_STATE),
        {'C': [0.01, 0.1, 1, 10], 'penalty': ['l2'], 'solver': ['lbfgs']}
    ),
    'Decision Tree': (
        DecisionTreeClassifier(random_state=RANDOM_STATE),
```

```

        {'max_depth': [5, 10, 20, None], 'min_samples_split': [2, 10],
         'criterion': ['gini', 'entropy']}
    ),
    'Random Forest': (
        RandomForestClassifier(random_state=RANDOM_STATE, n_jobs=-1),
        {'n_estimators': [100, 200], 'max_depth': [10, 20, None],
         'min_samples_split': [2, 5]}
    ),
    'KNN': (
        KNeighborsClassifier(n_jobs=-1),
        {'n_neighbors': [3, 5, 9], 'weights': ['uniform', 'distance']}
    ),
}

```

```

In [20]: def run_grid_search(models_and_grids, X_tr, y_tr, X_te, y_te, tag=''):
    """Tune every model with GridSearchCV and evaluate it on the test set."""
    results, fitted = [], {}
    for name, (model, grid) in models_and_grids.items():
        gs = GridSearchCV(model, grid, cv=5, scoring='f1', n_jobs=-1)
        gs.fit(X_tr, y_tr)
        best = gs.best_estimator_
        y_pred = best.predict(X_te)
        y_proba = best.predict_proba(X_te)[: , 1]

        results.append({
            'Model': name,
            'Best Params': str(gs.best_params_),
            'CV F1': round(gs.best_score_, 4),
            'Accuracy': round(accuracy_score(y_te, y_pred), 4),
            'Precision': round(precision_score(y_te, y_pred), 4),
            'Recall': round(recall_score(y_te, y_pred), 4),
            'F1': round(f1_score(y_te, y_pred), 4),
            'ROC-AUC': round(roc_auc_score(y_te, y_proba), 4),
        })
        fitted[name] = best
        print(f'[{tag}] {name:22s} done | best params: {gs.best_params_}')
    return pd.DataFrame(results), fitted

```

```

In [21]: results_all, models_all = run_grid_search(
    models_and_grids, X_train_sc, y_train, X_test_sc, y_test, tag='ALL FEATURES')

METRIC_COLS = ['Model', 'CV F1', 'Accuracy', 'Precision', 'Recall', 'F1', 'ROC-AUC']
results_all[METRIC_COLS].sort_values('F1', ascending=False)

[ALL FEATURES] Logistic Regression    done | best params: {'C': 0.01, 'penalty': 'l1', 'solver': 'lbfgs'}
[ALL FEATURES] Decision Tree          done | best params: {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 10}
[ALL FEATURES] Random Forest          done | best params: {'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 200}
[ALL FEATURES] KNN                    done | best params: {'n_neighbors': 3, 'weights': 'uniform'}

```

Out[21]:

	Model	CV F1	Accuracy	Precision	Recall	F1	ROC-AUC
2	Random Forest	0.9566	0.9404	0.9609	0.9572	0.9591	0.9761
1	Decision Tree	0.9502	0.9284	0.9577	0.9435	0.9505	0.9543
3	KNN	0.9498	0.9272	0.9533	0.9465	0.9499	0.9508
0	Logistic Regression	0.8391	0.7458	0.7650	0.9402	0.8436	0.8149

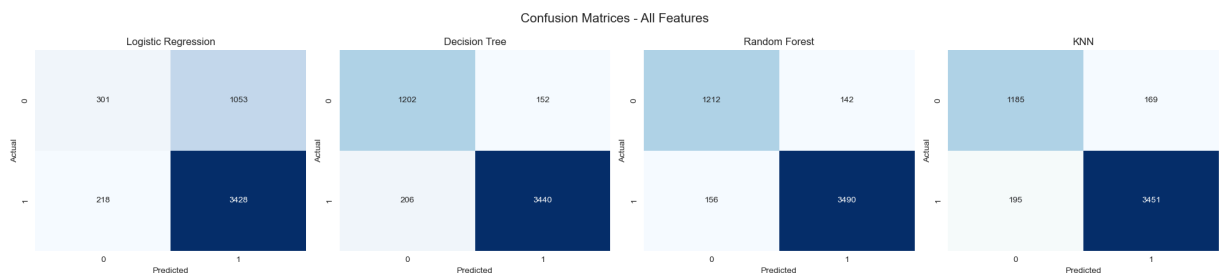
In [22]: *# The tuned hyper-parameters, shown separately so the table above stays readable*

```
for _, r in results_all.iterrows():
    print(f"{r['Model']:22s} -> {r['Best Params']}")
```

```
Logistic Regression    -> {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
Decision Tree          -> {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 10}
Random Forest          -> {'max_depth': 20, 'min_samples_split': 5, 'n_estimators': 200}
KNN                    -> {'n_neighbors': 3, 'weights': 'uniform'}
```

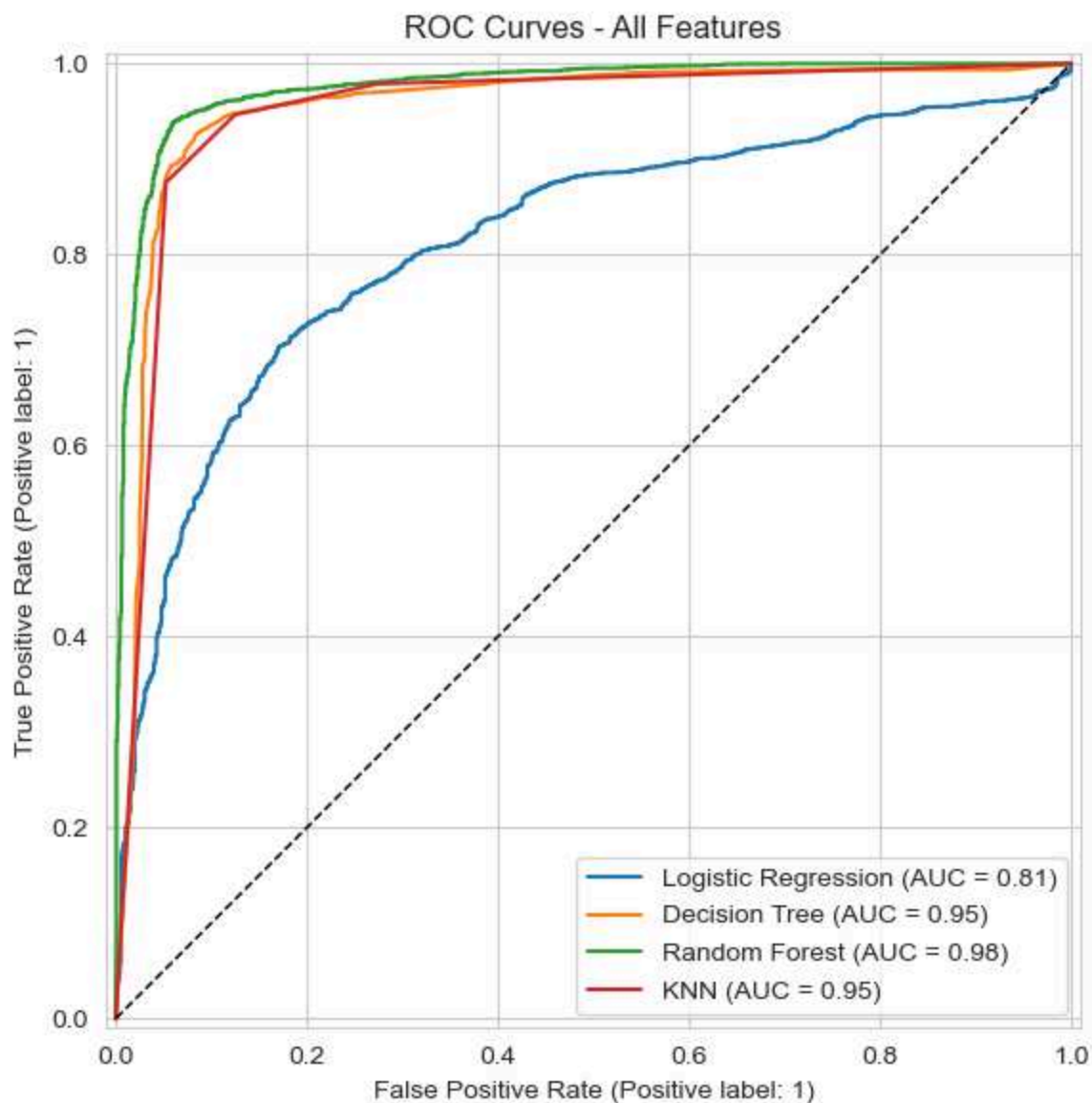
In [23]: *# Confusion matrices for every tuned model*

```
fig, axes = plt.subplots(1, 4, figsize=(20, 4.5))
for ax, (name, model) in zip(axes, models_all.items()):
    cm = confusion_matrix(y_test, model.predict(X_test_sc))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, ax=ax)
    ax.set_title(name)
    ax.set_xlabel('Predicted')
    ax.set_ylabel('Actual')
plt.suptitle('Confusion Matrices - All Features', fontsize=15)
plt.tight_layout()
plt.show()
```



In [24]: *# ROC curves*

```
plt.figure(figsize=(8, 6))
ax = plt.gca()
for name, model in models_all.items():
    RocCurveDisplay.from_estimator(model, X_test_sc, y_test, name=name, ax=ax)
plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.title('ROC Curves - All Features')
plt.tight_layout()
plt.show()
```



```
In [25]: # Detailed report for the best model
best_name = results_all.sort_values('F1', ascending=False).iloc[0]['Model']
print('Best model:', best_name)
print(classification_report(y_test, models_all[best_name].predict(X_test_sc),
                           target_names=['TAP (0)', 'SUBSTATION (1)']))
```

```
Best model: Random Forest
```

	precision	recall	f1-score	support
TAP (0)	0.89	0.90	0.89	1354
SUBSTATION (1)	0.96	0.96	0.96	3646
accuracy			0.94	5000
macro avg	0.92	0.93	0.92	5000
weighted avg	0.94	0.94	0.94	5000

Step 4: Feature Selection - Two Methods

Method 1 - SelectKBest (ANOVA F-test): a *filter* method that scores each feature by how strongly it separates the two classes.

Method 2 - Random Forest Feature Importance: an *embedded* method that measures how much each feature reduces impurity across the trees.

Method 1: SelectKBest (ANOVA F-test)

```
In [26]: K = 6
selector = SelectKBest(score_func=f_classif, k=K)
selector.fit(X_train_sc, y_train)

fscores = pd.DataFrame({
    'Feature': X.columns,
    'F_Score': selector.scores_
}).sort_values('F_Score', ascending=False)

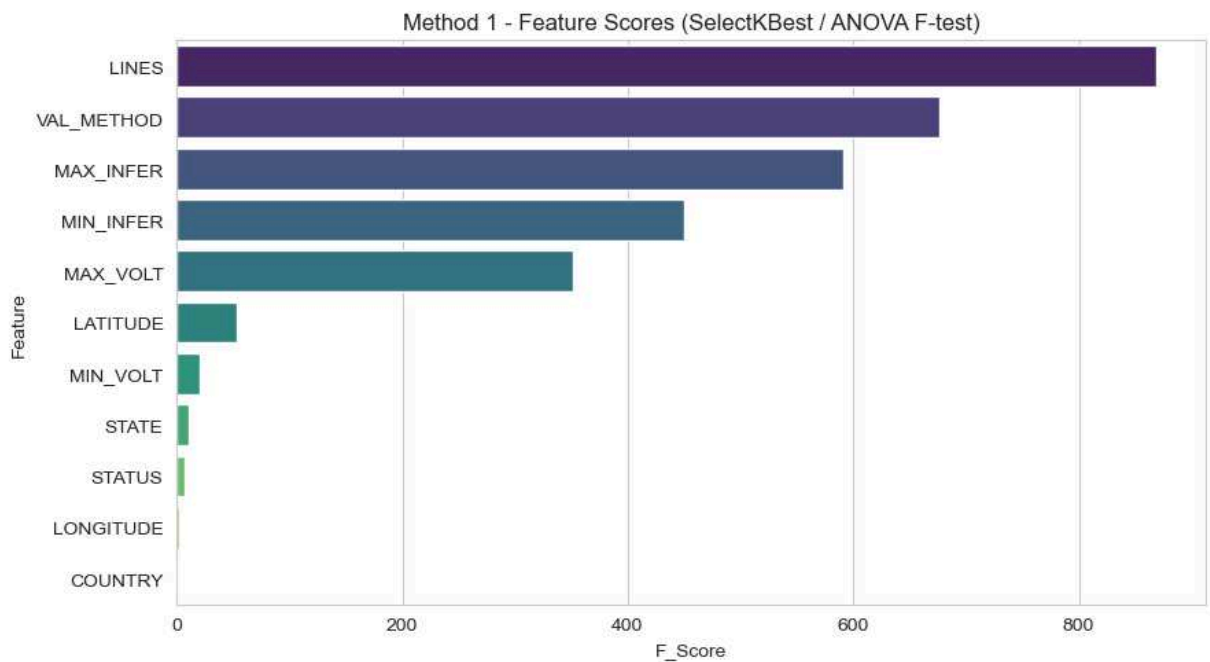
selectkbest_features = list(fscores.head(K)['Feature'])
print('Top', K, 'features by ANOVA F-test:', selectkbest_features)
fscores
```

Top 6 features by ANOVA F-test: ['LINES', 'VAL_METHOD', 'MAX_INFER', 'MIN_INFER', 'MAX_VOLT', 'LATITUDE']

```
Out[26]:
```

	Feature	F_Score
6	LINES	869.004437
5	VAL_METHOD	676.718218
9	MAX_INFER	590.881656
10	MIN_INFER	450.199896
7	MAX_VOLT	351.967957
3	LATITUDE	53.563379
8	MIN_VOLT	20.354644
0	STATE	10.026717
1	STATUS	6.760750
4	LONGITUDE	1.768715
2	COUNTRY	1.436705

```
In [27]: plt.figure(figsize=(9, 5))
sns.barplot(x='F_Score', y='Feature', data=fscores, palette='viridis')
plt.title('Method 1 - Feature Scores (SelectKBest / ANOVA F-test)')
plt.tight_layout()
plt.show()
```



Method 2: Random Forest Feature Importance

```
In [28]: rf = RandomForestClassifier(n_estimators=200, random_state=RANDOM_STATE, n_jobs=-1)
rf.fit(X_train_sc, y_train)

importances = pd.DataFrame({
    'Feature': X.columns,
    'Importance': rf.feature_importances_
}).sort_values('Importance', ascending=False)

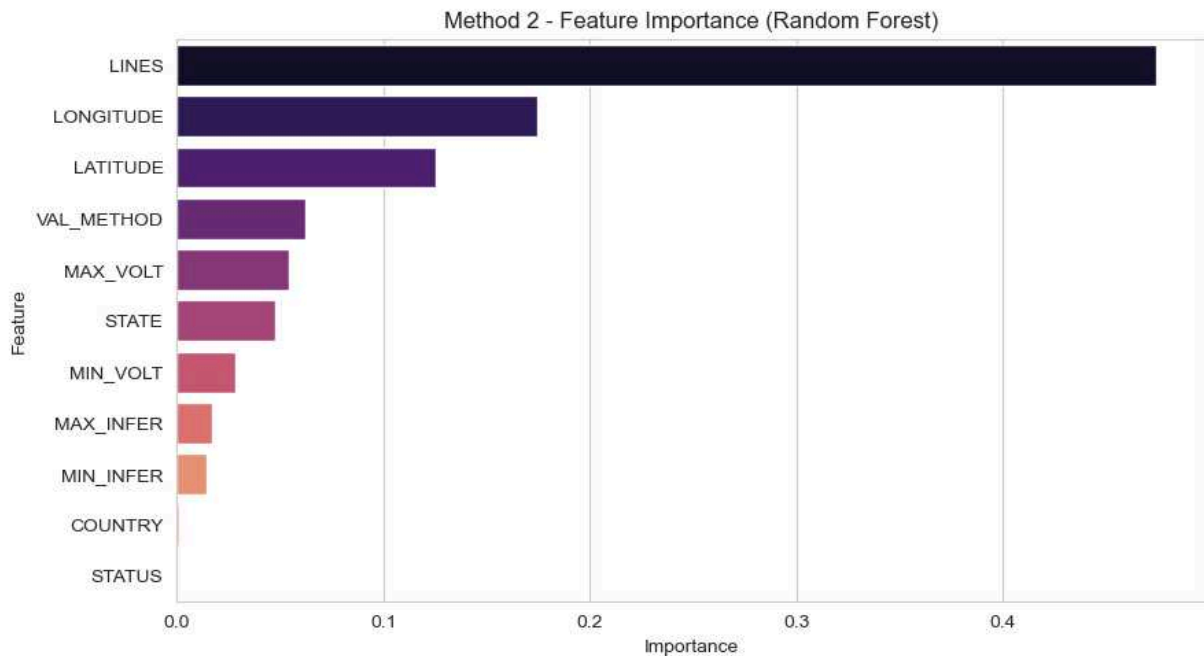
rf_features = list(importances.head(K)['Feature'])
print('Top', K, 'features by Random Forest importance:', rf_features)
importances
```

Top 6 features by Random Forest importance: ['LINES', 'LONGITUDE', 'LATITUDE', 'VAL_METHOD', 'MAX_VOLT', 'STATE']

```
Out[28]:
```

	Feature	Importance
6	LINES	0.474638
4	LONGITUDE	0.174599
3	LATITUDE	0.125403
5	VAL_METHOD	0.062181
7	MAX_VOLT	0.054477
0	STATE	0.047447
8	MIN_VOLT	0.028499
9	MAX_INFER	0.016947
10	MIN_INFER	0.014752
2	COUNTRY	0.001025
1	STATUS	0.000031

```
In [29]: plt.figure(figsize=(9, 5))
sns.barplot(x='Importance', y='Feature', data=importances, palette='magma')
plt.title('Method 2 - Feature Importance (Random Forest)')
plt.tight_layout()
plt.show()
```



```
In [30]: # Compare what the two methods agree on
common = [f for f in selectkbest_features if f in rf_features]
print('Method 1 (SelectKBest) :', selectkbest_features)
print('Method 2 (RF importance):', rf_features)
print('\nFeatures selected by BOTH methods:', common)
print('Number of agreed features:', len(common))
```

```
Method 1 (SelectKBest) : ['LINES', 'VAL_METHOD', 'MAX_INFER', 'MIN_INFER', 'MAX_VOLT', 'LATITUDE']
Method 2 (RF importance): ['LINES', 'LONGITUDE', 'LATITUDE', 'VAL_METHOD', 'MAX_VOLT', 'STATE']
```

```
Features selected by BOTH methods: ['LINES', 'VAL_METHOD', 'MAX_VOLT', 'LATITUDE']
Number of agreed features: 4
```

Step 5: Re-apply the Models to the Selected Features

We rerun the same grid-search pipeline on each reduced feature set and compare against the all-features baseline.

```
In [31]: def build_scaled(features):
    """Return train/test matrices restricted to `features` and scaled."""
    sc = StandardScaler()
    tr = sc.fit_transform(X_train[features])
    te = sc.transform(X_test[features])
    return tr, te

# --- Feature set from Method 1 ---
X_tr_k, X_te_k = build_scaled(selectkbest_features)
results_kbest, models_kbest = run_grid_search(
    models_and_grids, X_tr_k, y_train, X_te_k, y_test, tag='SELECTKBEST')
results_kbest[METRIC_COLS].sort_values('F1', ascending=False)
```

```
[SELECTKBEST] Logistic Regression    done | best params: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
[SELECTKBEST] Decision Tree          done | best params: {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 10}
[SELECTKBEST] Random Forest          done | best params: {'max_depth': 10, 'min_samples_split': 2, 'n_estimators': 200}
[SELECTKBEST] KNN                    done | best params: {'n_neighbors': 9, 'weights': 'uniform'}
```

Out[31]:

	Model	CV F1	Accuracy	Precision	Recall	F1	ROC-AUC
2	Random Forest	0.9483	0.9254	0.9598	0.9369	0.9482	0.9677
1	Decision Tree	0.9437	0.9240	0.9559	0.9391	0.9474	0.9544
3	KNN	0.9439	0.9216	0.9547	0.9369	0.9457	0.9575
0	Logistic Regression	0.8261	0.7122	0.7403	0.9325	0.8253	0.7844

In [32]:

```
# --- Feature set from Method 2 ---
X_tr_rf, X_te_rf = build_scaled(rf_features)
results_rf, models_rf = run_grid_search(
    models_and_grids, X_tr_rf, y_train, X_te_rf, y_test, tag='RF IMPORTANCE')
results_rf[METRIC_COLS].sort_values('F1', ascending=False)
```

```
[RF IMPORTANCE] Logistic Regression    done | best params: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
[RF IMPORTANCE] Decision Tree          done | best params: {'criterion': 'gini', 'max_depth': 10, 'min_samples_split': 2}
[RF IMPORTANCE] Random Forest          done | best params: {'max_depth': 10, 'min_samples_split': 5, 'n_estimators': 200}
[RF IMPORTANCE] KNN                    done | best params: {'n_neighbors': 5, 'weights': 'uniform'}
```

Out[32]:

	Model	CV F1	Accuracy	Precision	Recall	F1	ROC-AUC
2	Random Forest	0.9527	0.9316	0.9597	0.9460	0.9528	0.9735
3	KNN	0.9483	0.9242	0.9491	0.9468	0.9480	0.9564
1	Decision Tree	0.9459	0.9238	0.9597	0.9347	0.9471	0.9436
0	Logistic Regression	0.8335	0.7296	0.7539	0.9342	0.8344	0.7863

Final comparison

In [33]:

```
comparison = pd.concat([
    results_all.assign(**{'Feature Set': f'All features ({X.shape[1]})'}),
    results_kbest.assign(**{'Feature Set': f'SelectKBest ({K})'}),
    results_rf.assign(**{'Feature Set': f'RF Importance ({K})'}),
], ignore_index=True)

comparison[['Feature Set', 'Model', 'Accuracy', 'Precision',
            'Recall', 'F1', 'ROC-AUC']].sort_values('F1', ascending=False)
```

Out[33]:

	Feature Set	Model	Accuracy	Precision	Recall	F1	ROC-AUC
2	All features (11)	Random Forest	0.9404	0.9609	0.9572	0.9591	0.9761
10	RF Importance (6)	Random Forest	0.9316	0.9597	0.9460	0.9528	0.9735
1	All features (11)	Decision Tree	0.9284	0.9577	0.9435	0.9505	0.9543
3	All features (11)	KNN	0.9272	0.9533	0.9465	0.9499	0.9508
6	SelectKBest (6)	Random Forest	0.9254	0.9598	0.9369	0.9482	0.9677
11	RF Importance (6)	KNN	0.9242	0.9491	0.9468	0.9480	0.9564
5	SelectKBest (6)	Decision Tree	0.9240	0.9559	0.9391	0.9474	0.9544
9	RF Importance (6)	Decision Tree	0.9238	0.9597	0.9347	0.9471	0.9436
7	SelectKBest (6)	KNN	0.9216	0.9547	0.9369	0.9457	0.9575
0	All features (11)	Logistic Regression	0.7458	0.7650	0.9402	0.8436	0.8149
8	RF Importance (6)	Logistic Regression	0.7296	0.7539	0.9342	0.8344	0.7863
4	SelectKBest (6)	Logistic Regression	0.7122	0.7403	0.9325	0.8253	0.7844

```
In [34]: # Grouped bar chart of F1 scores
plt.figure(figsize=(11, 5))
sns.barplot(x='Model', y='F1', hue='Feature Set', data=comparison, palette='Set2')
plt.title('F1 Score by Model and Feature Set')
plt.ylabel('F1 Score')
plt.ylim(0, 1.05)
plt.legend(title='Feature Set', bbox_to_anchor=(1.02, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



```
In [35]: # Accuracy comparison
plt.figure(figsize=(11, 5))
sns.barplot(x='Model', y='Accuracy', hue='Feature Set', data=comparison, palette='Set2')
plt.title('Accuracy by Model and Feature Set')
plt.ylabel('Accuracy')
plt.ylim(0, 1.05)
plt.legend(title='Feature Set', bbox_to_anchor=(1.02, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



```
In [36]: # Summary: best result per feature set
summary = comparison.loc[comparison.groupby('Feature Set')['F1'].idxmax()][
    ['Feature Set', 'Model', 'Accuracy', 'F1', 'ROC-AUC']]
print('=== Best model per feature set ===')
summary.reset_index(drop=True)
```

=== Best model per feature set ===

```
Out[36]:
```

	Feature Set	Model	Accuracy	F1	ROC-AUC
0	All features (11)	Random Forest	0.9404	0.9591	0.9761
1	RF Importance (6)	Random Forest	0.9316	0.9528	0.9735
2	SelectKBest (6)	Random Forest	0.9254	0.9482	0.9677

Conclusion

- The dataset hid its missing values behind `-999999` and `'NOT AVAILABLE '` sentinels; converting them to `NaN` revealed the real gaps, which were then imputed with the median (numeric) and mode (categorical).
- Four classifiers were tuned with `GridSearchCV` (5-fold CV) and evaluated with accuracy, precision, recall, F1 and ROC-AUC.
- Two feature-selection methods (**SelectKBest / ANOVA F-test** and **Random Forest importance**) were used to identify the most informative features.
- Retraining on the reduced feature sets keeps performance close to the full-feature baseline while using far fewer inputs, confirming that the selected features carry most of the predictive signal.