DroneDetect V2 - Model Comparison (V3)

Comprehensive comparison of all trained models:

- **SVM** (PSD features)
- **VGG16** (Spectrogram features - RGB via Viridis colormap)
- **ResNet50** (Spectrogram features - RGB via Viridis colormap)
- **RF-UAV-Net** (Raw IQ features)

Evaluation metrics:

- Accuracy, Precision, Recall, F1-Score
- Confusion matrices
- Statistical significance tests (McNemar, Bootstrap CI, Cohen's kappa)
- **Per-sample inference time with proper benchmarking**
- Model size comparison
- Error analysis
- **Sample size comparison**
- **Test sample export to GDrive**

Inference timing methodology:

- Per-sample timing with warm-up (10 iterations)
- 100 timing runs per model
- Reports p50, p95, p99 latencies (median recommended for comparison)
- GPU synchronization for accurate CUDA timing
- Uses `time.perf_counter()` for high-resolution measurements

1. Mount Google Drive

In [1]:
```
from google.colab import drive

drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=Tru
e).
```

2. Imports

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as tv_models
from torch.utils.data import Dataset, DataLoader, TensorDataset
from sklearn.model_selection import StratifiedGroupKFold
from sklearn.metrics import (
    confusion_matrix, classification_report, accuracy_score,
    f1_score, precision_recall_fscore_support, cohen_kappa_score
)
from sklearn.svm import SVC
from scipy.stats import chi2
import pickle
import os
import gc
import time
import re
from pathlib import Path

plt.style.use('seaborn-v0_8-darkgrid')
%matplotlib inline

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
print(f"Using device: {device}")

# Figure saving configuration
NOTEBOOK_NAME = "model_comparison_COLAB_V3"
FIGURES_DIR = Path("figures") / NOTEBOOK_NAME


def save_figure(fig) -> None:
    """Save plotly figure to PNG file using the figure's title as filename."""
    FIGURES_DIR.mkdir(parents=True, exist_ok=True)
    title = fig.layout.title.text if fig.layout.title.text else "untitled"
    filename = re.sub(r'[^\w\s-]', '', title).strip()
    filename = re.sub(r'[\s-]+', '_', filename)
    filepath = FIGURES_DIR / f"{filename}.png"
    try:
        fig.write_image(str(filepath), width=1200, height=800)
        print(f"Saved: {filepath}")
    except Exception as e:
        print(f"Warning: Could not save figure (kaleido required): {e}")
```

```
Using device: cuda
```

3. Configuration

In [3]:

```
CONFIG = {
    # Paths
    'psd_path': 'drive/MyDrive/DroneDetect_V2/output/features/psd_features.npz',
    'spectrogram_path': 'drive/MyDrive/DroneDetect_V2/output/features/spectrogram_features.npz',
    'iq_path': 'drive/MyDrive/DroneDetect_V2/output/features/iq_features.npz',
    'models_dir': 'drive/MyDrive/DroneDetect_V2/output/models/',

    # Sample export path
    'sample_export_dir': 'drive/MyDrive/DroneDetect_V2/output/test_samples/',

    # Split parameters
    'test_size': 0.2,
    'random_state': 42,

    # Evaluation parameters
    'batch_size': 128,
```

```
    'n_bootstrap': 1000,
    'n_timing_runs': 100,
    'warmup_runs': 10,

    # Device
    'device': device,

    # Visualization colors
    'colors': {
        'SVM': '#3498db',
        'VGG16': '#e74c3c',
        'ResNet50': '#2ecc71',
        'RFUAVNet': '#9b59b6'
    }
}

# Sample sizes for inference comparison and export
SAMPLE_SIZES = [20, 100, 500, 1000, 1320]

print(f"Configuration: {CONFIG}")
print(f"Sample sizes to evaluate/export: {SAMPLE_SIZES}")
```

```
Configuration: {'psd_path': 'drive/MyDrive/DroneDetect_V2/output/features/psd_features.npz', 'spectrogram_path': 'drive/MyDriv
e/DroneDetect_V2/output/features/spectrogram_features.npz', 'iq_path': 'drive/MyDrive/DroneDetect_V2/output/features/iq_feature
s.npz', 'models_dir': 'drive/MyDrive/DroneDetect_V2/output/models/', 'sample_export_dir': 'drive/MyDrive/DroneDetect_V2/output/
test_samples/', 'test_size': 0.2, 'random_state': 42, 'batch_size': 128, 'n_bootstrap': 1000, 'n_timing_runs': 100, 'warmup_run
s': 10, 'device': device(type='cuda'), 'colors': {'SVM': '#3498db', 'VGG16': '#e74c3c', 'ResNet50': '#2ecc71', 'RFUAVNet': '#9b
59b6'}}
Sample sizes to evaluate/export: [20, 100, 500, 1000, 1320]
```

4. Helper Functions - Model Definitions

All model classes defined once (with `.reshape()` fix for non-contiguous tensors).

Note: Spectrograms are already RGB (3 channels via Viridis colormap from preprocessing). No grayscale-to-RGB conversion needed.

In [4]:
```
class VGG16FC(nn.Module):
    """VGG16 with frozen features and trainable classifier."""
```

```python
    def __init__(self, num_classes: int, from_array: bool = False):
        super().__init__()
        self.from_array = from_array

        vgg = tv_models.vgg16(weights='IMAGENET1K_V1')
        self.features = nn.Sequential(*list(vgg.children())[:-1])

        for param in self.features.parameters():
            param.requires_grad = False

        self.classifier = nn.Linear(25088, num_classes)

    def forward(self, x):
        if self.from_array:
            x = x.unsqueeze(1).repeat(1, 3, 1, 1)
        elif x.dim() == 4 and x.shape[-1] == 3:
            x = x.permute(0, 3, 1, 2)

        x = self.features(x)
        x = x.reshape(x.size(0), -1)  # Use reshape instead of view
        return self.classifier(x)


class ResNet50FC(nn.Module):
    """ResNet50 with frozen features and trainable classifier."""

    def __init__(self, num_classes: int):
        super().__init__()

        resnet = tv_models.resnet50(weights='IMAGENET1K_V1')
        self.features = nn.Sequential(*list(resnet.children())[:-2])

        for param in self.features.parameters():
            param.requires_grad = False

        self.classifier = nn.Linear(100352, num_classes)

    def forward(self, x):
        if x.dim() == 4 and x.shape[-1] == 3:
            x = x.permute(0, 3, 1, 2)
```

```
        x = self.features(x)
        x = x.reshape(x.size(0), -1)  # Use reshape instead of view
        return self.classifier(x)


class RFUAVNet(nn.Module):
    """RF-UAV-Net: 1D CNN for raw IQ classification."""

    def __init__(self, num_classes: int):
        super().__init__()

        # R-unit
        self.conv_r = nn.Conv1d(2, 64, kernel_size=5, stride=5)
        self.bn_r = nn.BatchNorm1d(64)
        self.elu_r = nn.ELU()

        # G-units (4x)
        self.g_convs = nn.ModuleList([
            nn.Conv1d(64, 64, kernel_size=3, stride=2, groups=8)
            for _ in range(4)
        ])
        self.g_bns = nn.ModuleList([nn.BatchNorm1d(64) for _ in range(4)])
        self.g_elus = nn.ModuleList([nn.ELU() for _ in range(4)])

        self.pool = nn.MaxPool1d(kernel_size=2, stride=2)

        # Multi-scale GAP
        self.gap1000 = nn.AvgPool1d(1000)
        self.gap500 = nn.AvgPool1d(500)
        self.gap250 = nn.AvgPool1d(250)
        self.gap125 = nn.AvgPool1d(125)

        # Classifier
        self.fc = nn.Linear(320, num_classes)

    def forward(self, x):
        # R-unit
        x = self.elu_r(self.bn_r(self.conv_r(x)))

        # G-units with residual connections
        g_outputs = []
```

```
        for i in range(4):
            g_out = self.g_elus[i](self.g_bns[i](self.g_convs[i](F.pad(x, (1, 0)))))
            g_outputs.append(g_out)
            x = g_out + self.pool(x)

        # Multi-scale GAP
        gaps = [
            self.gap1000(g_outputs[0]),
            self.gap500(g_outputs[1]),
            self.gap250(g_outputs[2]),
            self.gap125(g_outputs[3]),
            self.gap125(x)
        ]

        x = torch.cat(gaps, dim=1).flatten(start_dim=1)
        return self.fc(x)


class SpectrogramDataset(Dataset):
    """Dataset for RGB spectrograms (already 3 channels from preprocessing)."""

    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        x = torch.from_numpy(self.X[idx]).float()
        y = torch.tensor(self.y[idx]).long()
        return x, y


print("Model classes defined (VGG16FC, ResNet50FC, RFUAVNet, SpectrogramDataset)")
```

```
Model classes defined (VGG16FC, ResNet50FC, RFUAVNet, SpectrogramDataset)
```

5. Helper Functions - Split and Evaluation

In [5]:

```python
def get_stratified_file_split(X, y, file_ids, test_size=0.2, random_state=42):
    """
    Split data at FILE level to prevent data leakage.

    Parameters
    ----------
    X : array-like
        Features (n_samples, ...)
    y : array-like
        Labels for stratification (n_samples,)
    file_ids : array-like
        Source file ID for each sample (n_samples,)
    test_size : float
        Approximate test set proportion
    random_state : int
        Random seed for reproducibility

    Returns
    -------
    train_idx, test_idx : arrays
        Indices for train/test split
    """
    n_splits = int(1 / test_size)
    sgkf = StratifiedGroupKFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    train_idx, test_idx = next(sgkf.split(X, y, groups=file_ids))

    # Verify no file leakage
    train_files = set(file_ids[train_idx])
    test_files = set(file_ids[test_idx])
    assert len(train_files & test_files) == 0, "Data leakage detected"

    return train_idx, test_idx


def get_sample_indices(y, n_samples, random_state=42):
    """
    Get stratified sample indices.

    Parameters
    ----------
```

```
    y : array-like
        Labels for stratification
    n_samples : int or None
        Number of samples (None = all)
    random_state : int
        Random seed

    Returns
    -------
    indices : array
        Sample indices
    """
    if n_samples is None or n_samples >= len(y):
        return np.arange(len(y))

    np.random.seed(random_state)
    unique_classes = np.unique(y)
    indices = []
    samples_per_class = n_samples // len(unique_classes)

    for cls in unique_classes:
        cls_indices = np.where(y == cls)[0]
        n_take = min(samples_per_class, len(cls_indices))
        chosen = np.random.choice(cls_indices, size=n_take, replace=False)
        indices.extend(chosen)

    return np.array(indices)


def evaluate_pytorch_model(model, dataloader, device, n_timing_runs=100, warmup_runs=10):
    """
    Evaluate PyTorch model with per-sample inference timing.

    Parameters
    ----------
    model : torch.nn.Module
        Model to evaluate
    dataloader : DataLoader
        Test data loader
    device : torch.device
        Device to run on
```

```
    n_timing_runs : int
        Number of timing iterations
    warmup_runs : int
        Number of warmup iterations

    Returns
    -------
    predictions : array
        Model predictions
    labels : array
        True labels
    timing_stats : dict
        Timing statistics with p50_ms, p95_ms, p99_ms, mean_ms, std_ms
    """
    model.eval()
    all_preds = []
    all_labels = []

    # First pass: get all predictions
    with torch.no_grad():
        for batch_x, batch_y in dataloader:
            batch_x = batch_x.to(device)
            outputs = model(batch_x)
            _, predicted = torch.max(outputs, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(batch_y.numpy())

    # Get a single sample for timing
    sample_x, _ = next(iter(dataloader))
    single_sample = sample_x[:1].to(device)

    # Warmup runs
    with torch.no_grad():
        for _ in range(warmup_runs):
            _ = model(single_sample)

    # Timing runs
    if torch.cuda.is_available():
        torch.cuda.synchronize()

    times = []
```

```
    with torch.no_grad():
        for _ in range(n_timing_runs):
            if torch.cuda.is_available():
                torch.cuda.synchronize()
            start = time.perf_counter()
            _ = model(single_sample)
            if torch.cuda.is_available():
                torch.cuda.synchronize()
            end = time.perf_counter()
            times.append((end - start) * 1000)  # Convert to ms

    times = np.array(times)
    timing_stats = {
        'p50_ms': np.percentile(times, 50),
        'p95_ms': np.percentile(times, 95),
        'p99_ms': np.percentile(times, 99),
        'mean_ms': np.mean(times),
        'std_ms': np.std(times)
    }

    return np.array(all_preds), np.array(all_labels), timing_stats


def benchmark_svm_model(model, X, n_timing_runs=100, warmup_runs=10):
    """
    Benchmark SVM model with per-sample inference timing.

    Parameters
    ----------
    model : sklearn estimator
        Trained SVM model
    X : array-like
        Test features
    n_timing_runs : int
        Number of timing iterations
    warmup_runs : int
        Number of warmup iterations

    Returns
    -------
    timing_stats : dict
```

```
        Timing statistics with p50_ms, p95_ms, p99_ms, mean_ms, std_ms
    """
    single_sample = X[:1]

    # Warmup runs
    for _ in range(warmup_runs):
        _ = model.predict(single_sample)

    # Timing runs
    times = []
    for _ in range(n_timing_runs):
        start = time.perf_counter()
        _ = model.predict(single_sample)
        end = time.perf_counter()
        times.append((end - start) * 1000)  # Convert to ms

    times = np.array(times)
    return {
        'p50_ms': np.percentile(times, 50),
        'p95_ms': np.percentile(times, 95),
        'p99_ms': np.percentile(times, 99),
        'mean_ms': np.mean(times),
        'std_ms': np.std(times)
    }


def get_model_size_mb(model_path):
    """Get model file size in MB."""
    size_bytes = os.path.getsize(model_path)
    return size_bytes / (1024 * 1024)


print("Helper functions defined")
```

```
Helper functions defined
```

6. Load Test Data

Load all feature types with file-level split (no data leakage).

In [6]:
```
# Load PSD features
data_psd = np.load(CONFIG['psd_path'], allow_pickle=True)

X_psd = data_psd['X']
y_drone_psd = data_psd['y_drone']
y_interference_psd = data_psd['y_interference']
file_ids_psd = data_psd['file_ids']
drone_classes = data_psd['drone_classes']
interference_classes = data_psd['interference_classes']

# File-level split
_, test_idx_psd = get_stratified_file_split(
    X_psd, y_drone_psd, file_ids_psd,
    test_size=CONFIG['test_size'],
    random_state=CONFIG['random_state']
)

X_test_psd = X_psd[test_idx_psd]
y_test_psd = y_drone_psd[test_idx_psd]
y_test_interference_psd = y_interference_psd[test_idx_psd]

print(f"PSD test set: {X_test_psd.shape}")
print(f"Drone classes: {drone_classes}")
print(f"Interference classes: {interference_classes}")

# Cleanup
del X_psd, y_drone_psd, file_ids_psd, data_psd
gc.collect()
```

```
PSD test set: (3891, 1024)
Drone classes: ['AIR' 'DIS' 'INS' 'MA1' 'MAV' 'MIN' 'PHA']
Interference classes: ['BOTH' 'CLEAN']
```

Out[6]: 3185

In [7]:
```
# Load spectrogram features (memory-mapped)
data_spec = np.load(CONFIG['spectrogram_path'], mmap_mode='r')

X_spec = data_spec['X']
y_drone_spec = data_spec['y_drone']
y_interference_spec = data_spec['y_interference']
```

```
file_ids_spec = data_spec['file_ids']

# File-level split
_, test_idx_spec = get_stratified_file_split(
    X_spec, y_drone_spec, file_ids_spec,
    test_size=CONFIG['test_size'],
    random_state=CONFIG['random_state']
)

X_test_spec = X_spec[test_idx_spec]
y_test_spec = y_drone_spec[test_idx_spec]
y_test_interference_spec = y_interference_spec[test_idx_spec]

print(f"Spectrogram test set (RGB): {X_test_spec.shape}")

# Create dataset and dataloader
test_dataset_spec = SpectrogramDataset(X_test_spec, y_test_spec)
test_loader_spec = DataLoader(
    test_dataset_spec,
    batch_size=CONFIG['batch_size'],
    shuffle=False,
    num_workers=0,
    pin_memory=True if torch.cuda.is_available() else False
)

print(f"Test batches: {len(test_loader_spec)}")

# Cleanup
del X_spec, y_drone_spec, file_ids_spec, data_spec, test_idx_spec
gc.collect()
```

```
Spectrogram test set (RGB): (3891, 224, 224, 3)
Test batches: 31
```

Out[7]: 44

In [8]:
```
# Load IQ features (memory-mapped)
data_iq = np.load(CONFIG['iq_path'], mmap_mode='r')

X_iq = data_iq['X']
y_drone_iq = data_iq['y_drone']
```

```
y_interference_iq = data_iq['y_interference']
file_ids_iq = data_iq['file_ids']

# File-level split
_, test_idx_iq = get_stratified_file_split(
    X_iq, y_drone_iq, file_ids_iq,
    test_size=CONFIG['test_size'],
    random_state=CONFIG['random_state']
)

X_test_iq = X_iq[test_idx_iq]
y_test_iq = y_drone_iq[test_idx_iq]
y_test_interference_iq = y_interference_iq[test_idx_iq]

print(f"IQ test set: {X_test_iq.shape}")

# Convert to PyTorch tensors
X_test_iq_t = torch.from_numpy(X_test_iq.copy()).float()
y_test_iq_t = torch.from_numpy(y_test_iq.copy()).long()

# Create dataloader
test_dataset_iq = TensorDataset(X_test_iq_t, y_test_iq_t)
test_loader_iq = DataLoader(
    test_dataset_iq,
    batch_size=CONFIG['batch_size'],
    shuffle=False
)

print(f"Test batches: {len(test_loader_iq)}")

# Cleanup
del X_iq, y_drone_iq, file_ids_iq, data_iq, test_idx_iq
gc.collect()
```

```
IQ test set: (3891, 2, 10000)
Test batches: 31
```

Out[8]: 0

In [9]:
```
# Verify all test sets have same labels
print(f"PSD test labels: {len(y_test_psd)} samples")
```

```
print(f"Spectrogram test labels: {len(y_test_spec)} samples")
print(f"IQ test labels: {len(y_test_iq_t)} samples")

# Check label distribution
unique, counts = np.unique(y_test_psd, return_counts=True)
print("\nTest set class distribution:")
for cls, count in zip(drone_classes, counts):
    print(f"  {cls}: {count} samples")
```

```
PSD test labels: 3891 samples
Spectrogram test labels: 3891 samples
IQ test labels: 3891 samples

Test set class distribution:
  AIR: 600 samples
  DIS: 400 samples
  INS: 591 samples
  MA1: 600 samples
  MAV: 700 samples
  MIN: 600 samples
  PHA: 400 samples
```

7. Model Evaluation

In [10]:

```
# Load and evaluate SVM
svm_path = os.path.join(CONFIG['models_dir'], 'svm_psd_drone.pkl')
with open(svm_path, 'rb') as f:
    svm_data = pickle.load(f)

svm_model = svm_data['model']

# Full test set predictions
print("Inferring on full test set...")
svm_preds = svm_model.predict(X_test_psd)

svm_acc = accuracy_score(y_test_psd, svm_preds)
svm_f1 = f1_score(y_test_psd, svm_preds, average='weighted')

print(f"SVM Test Accuracy: {svm_acc:.4f}")
print(f"SVM Test F1-Score: {svm_f1:.4f}")
```

```
# Benchmark single sample inference
print("\nBenchmarking single-sample inference...")
svm_timing = benchmark_svm_model(
    svm_model, X_test_psd,
    n_timing_runs=CONFIG['n_timing_runs'],
    warmup_runs=CONFIG['warmup_runs']
)
print(f"Inference time per sample: {svm_timing['p50_ms']:.3f} ms (p50), {svm_timing['p95_ms']:.3f} ms (p95)")
print(f"SVM Model Size: {get_model_size_mb(svm_path):.2f} MB")
```

```
Inferring on full test set...
SVM Test Accuracy: 0.8293
SVM Test F1-Score: 0.8277

Benchmarking single-sample inference...
Inference time per sample: 8.803 ms (p50), 9.055 ms (p95)
SVM Model Size: 53.83 MB
```

In [11]:

```
# Load and evaluate VGG16
vgg_path = os.path.join(CONFIG['models_dir'], 'vgg16_cnn.pth')
vgg_checkpoint = torch.load(vgg_path, map_location=device, weights_only=False)

num_classes = len(drone_classes)
vgg_model = VGG16FC(num_classes=num_classes)
vgg_model.load_state_dict(vgg_checkpoint['model_state_dict'])
vgg_model = vgg_model.to(device)

vgg_preds, vgg_labels, vgg_timing = evaluate_pytorch_model(
    vgg_model, test_loader_spec, device,
    n_timing_runs=CONFIG['n_timing_runs'],
    warmup_runs=CONFIG['warmup_runs']
)

vgg_acc = accuracy_score(vgg_labels, vgg_preds)
vgg_f1 = f1_score(vgg_labels, vgg_preds, average='weighted')

print(f"VGG16 Test Accuracy: {vgg_acc:.4f}")
print(f"VGG16 Test F1-Score: {vgg_f1:.4f}")
print(f"VGG16 Inference Time per sample: {vgg_timing['p50_ms']:.3f} ms (p50), {vgg_timing['p95_ms']:.3f} ms (p95)")
print(f"VGG16 Model Size: {get_model_size_mb(vgg_path):.2f} MB")
```

```
del vgg_model, vgg_checkpoint
gc.collect()
if torch.cuda.is_available():
    torch.cuda.empty_cache()
```

```
VGG16 Test Accuracy: 0.7851
VGG16 Test F1-Score: 0.7839
VGG16 Inference Time per sample: 1.622 ms (p50), 2.046 ms (p95)
VGG16 Model Size: 56.81 MB
```

In [12]:

```
# Load and evaluate ResNet50
resnet_path = os.path.join(CONFIG['models_dir'], 'resnet50_cnn.pth')
resnet_checkpoint = torch.load(resnet_path, map_location=device, weights_only=False)

resnet_model = ResNet50FC(num_classes=len(drone_classes))
resnet_model.load_state_dict(resnet_checkpoint['model_state_dict'])
resnet_model = resnet_model.to(device)

resnet_preds, resnet_labels, resnet_timing = evaluate_pytorch_model(
    resnet_model, test_loader_spec, device,
    n_timing_runs=CONFIG['n_timing_runs'],
    warmup_runs=CONFIG['warmup_runs']
)

resnet_acc = accuracy_score(resnet_labels, resnet_preds)
resnet_f1 = f1_score(resnet_labels, resnet_preds, average='weighted')

print(f"ResNet50 Test Accuracy: {resnet_acc:.4f}")
print(f"ResNet50 Test F1-Score: {resnet_f1:.4f}")
print(f"ResNet50 Inference Time per sample: {resnet_timing['p50_ms']:.3f} ms (p50), {resnet_timing['p95_ms']:.3f} ms (p95)")
print(f"ResNet50 Model Size: {get_model_size_mb(resnet_path):.2f} MB")

del resnet_model, resnet_checkpoint
gc.collect()
if torch.cuda.is_available():
    torch.cuda.empty_cache()
```

```
ResNet50 Test Accuracy: 0.7368
ResNet50 Test F1-Score: 0.7315
ResNet50 Inference Time per sample: 6.777 ms (p50), 7.360 ms (p95)
ResNet50 Model Size: 92.66 MB
```

In [13]:
```python
# Load and evaluate RF-UAV-Net
rfuavnet_path = os.path.join(CONFIG['models_dir'], 'rfuavnet_iq.pth')
rfuavnet_checkpoint = torch.load(rfuavnet_path, map_location=device, weights_only=False)

rfuavnet_model = RFUAVNet(num_classes=num_classes)
rfuavnet_model.load_state_dict(rfuavnet_checkpoint['model_state_dict'])
rfuavnet_model = rfuavnet_model.to(device)

rfuavnet_preds, rfuavnet_labels, rfuavnet_timing = evaluate_pytorch_model(
    rfuavnet_model, test_loader_iq, device,
    n_timing_runs=CONFIG['n_timing_runs'],
    warmup_runs=CONFIG['warmup_runs']
)

rfuavnet_acc = accuracy_score(rfuavnet_labels, rfuavnet_preds)
rfuavnet_f1 = f1_score(rfuavnet_labels, rfuavnet_preds, average='weighted')

print(f"RF-UAV-Net Test Accuracy: {rfuavnet_acc:.4f}")
print(f"RF-UAV-Net Test F1-Score: {rfuavnet_f1:.4f}")
print(f"RF-UAV-Net Inference Time per sample: {rfuavnet_timing['p50_ms']:.3f} ms (p50), {rfuavnet_timing['p95_ms']:.3f} ms (p9
print(f"RF-UAV-Net Model Size: {get_model_size_mb(rfuavnet_path):.2f} MB")

del rfuavnet_model, rfuavnet_checkpoint
gc.collect()
if torch.cuda.is_available():
    torch.cuda.empty_cache()
```

```
RF-UAV-Net Test Accuracy: 0.5289
RF-UAV-Net Test F1-Score: 0.5239
RF-UAV-Net Inference Time per sample: 1.487 ms (p50), 1.521 ms (p95)
RF-UAV-Net Model Size: 0.06 MB
```

8. Aggregate Results

In [14]:

```python
# Create results DataFrame
results_df = pd.DataFrame({
    'Model': ['SVM', 'VGG16', 'ResNet50', 'RFUAVNet'],
    'Features': ['PSD', 'Spectrogram', 'Spectrogram', 'Raw IQ'],
    'Accuracy': [svm_acc, vgg_acc, resnet_acc, rfuavnet_acc],
    'F1-Score': [svm_f1, vgg_f1, resnet_f1, rfuavnet_f1],
    'Inference_p50_ms': [
        svm_timing['p50_ms'],
        vgg_timing['p50_ms'],
        resnet_timing['p50_ms'],
        rfuavnet_timing['p50_ms']
    ],
    'Inference_p95_ms': [
        svm_timing['p95_ms'],
        vgg_timing['p95_ms'],
        resnet_timing['p95_ms'],
        rfuavnet_timing['p95_ms']
    ],
    'Inference_p99_ms': [
        svm_timing['p99_ms'],
        vgg_timing['p99_ms'],
        resnet_timing['p99_ms'],
        rfuavnet_timing['p99_ms']
    ],
    'Model_Size_MB': [
        get_model_size_mb(svm_path),
        get_model_size_mb(vgg_path),
        get_model_size_mb(resnet_path),
        get_model_size_mb(rfuavnet_path)
    ]
})

results_df = results_df.sort_values('Accuracy', ascending=False).reset_index(drop=True)

print("\n=== Model Comparison Results ===")
print(results_df.to_string(index=False))

# Store predictions for statistical tests
predictions = {
    'SVM': svm_preds,
```

```
    'VGG16': vgg_preds,
    'ResNet50': resnet_preds,
    'RFUAVNet': rfuavnet_preds
}
```

```
=== Model Comparison Results ===
   Model    Features  Accuracy  F1-Score  Inference_p50_ms  Inference_p95_ms  Inference_p99_ms  Model_Size_MB
     SVM         PSD  0.829350  0.827694          8.802722          9.054884          9.348470      53.825613
   VGG16 Spectrogram  0.785145  0.783906          1.622480          2.046268          2.116915      56.813128
ResNet50 Spectrogram  0.736829  0.731542          6.777392          7.359705          8.552156      92.663236
RFUAVNet      Raw IQ  0.528913  0.523886          1.487095          1.521441          1.557167       0.058868
```

10. Per-Class Metrics

In [15]:

```python
# Calculate per-class metrics
per_class_metrics = {}

for model_name, preds in predictions.items():
    labels = y_test_psd
    precision, recall, f1, support = precision_recall_fscore_support(
        labels, preds, labels=range(len(drone_classes)), zero_division=0
    )
    per_class_metrics[model_name] = {
        'precision': precision,
        'recall': recall,
        'f1': f1,
        'support': support
    }

# Print per-class F1 for best model
best_model = results_df.iloc[0]['Model']
print(f"\n=== {best_model} Per-Class Performance ===")
print(f"{'Class':<10} {'Precision':<12} {'Recall':<12} {'F1-Score':<12} {'Support':<10}")
print("-" * 60)
for i, cls in enumerate(drone_classes):
    metrics = per_class_metrics[best_model]
    print(f"{cls:<10} {metrics['precision'][i]:<12.4f} {metrics['recall'][i]:<12.4f} "
          f"{metrics['f1'][i]:<12.4f} {metrics['support'][i]:<10}")
```

```
=== SVM Per-Class Performance ===
Class      Precision    Recall       F1-Score     Support
------------------------------------------------------------
AIR        0.6706       0.7500       0.7081       600
DIS        1.0000       0.9575       0.9783       400
INS        0.8758       0.9780       0.9241       591
MA1        0.6439       0.5967       0.6194       600
MAV        0.7828       0.6900       0.7335       700
MIN        1.0000       0.9967       0.9983       600
PHA        0.9286       0.9425       0.9355       400
```

11. Visualizations

In [16]:
```python
# Accuracy & F1 Comparison with Plotly
fig = make_subplots(rows=1, cols=2, subplot_titles=('Model Accuracy Comparison', 'Model F1-Score Comparison'))

models = results_df['Model'].values
colors = [CONFIG['colors'][m] for m in models]

# Accuracy bars
fig.add_trace(go.Bar(
    x=list(models), y=results_df['Accuracy'].values,
    marker_color=colors, text=[f"{v:.4f}" for v in results_df['Accuracy'].values],
    textposition='outside', name='Accuracy'
), row=1, col=1)

# F1-Score bars
fig.add_trace(go.Bar(
    x=list(models), y=results_df['F1-Score'].values,
    marker_color=colors, text=[f"{v:.4f}" for v in results_df['F1-Score'].values],
    textposition='outside', name='F1-Score', showlegend=False
), row=1, col=2)

fig.update_layout(
    title='Model Performance Comparison - Accuracy and F1-Score',
    height=500, width=1200
)
fig.update_yaxes(range=[0, 1.1], title_text='Accuracy', row=1, col=1)
fig.update_yaxes(range=[0, 1.1], title_text='F1-Score (Weighted)', row=1, col=2)
```

```
fig.show()
save_figure(fig)
```

```
Warning: Could not save figure (kaleido required):
Image export using the "kaleido" engine requires the kaleido package,
which can be installed using pip:
    $ pip install -U kaleido
```

In [17]:

```python
# Confusion Matrices with Plotly
fig = make_subplots(rows=2, cols=2,
                    subplot_titles=['SVM', 'VGG16', 'ResNet50', 'RFUAVNet'],
                    horizontal_spacing=0.12, vertical_spacing=0.12)

colorscales = ['Blues', 'Reds', 'Greens', 'Purples']
model_names = ['SVM', 'VGG16', 'ResNet50', 'RFUAVNet']
positions = [(1, 1), (1, 2), (2, 1), (2, 2)]

for idx, (model_name, (row, col)) in enumerate(zip(model_names, positions)):
    cm = confusion_matrix(y_test_psd, predictions[model_name])
    acc = results_df[results_df['Model'] == model_name]['Accuracy'].values[0]

    fig.add_trace(go.Heatmap(
        z=cm, x=list(drone_classes), y=list(drone_classes),
        colorscale=colorscales[idx], text=cm, texttemplate='%{text}',
        textfont={'size': 10}, showscale=False, hoverongaps=False
    ), row=row, col=col)

fig.update_layout(
    title='All Models Confusion Matrices',
    height=900, width=1000
)

# Update axes for all subplots
for i in range(1, 3):
    for j in range(1, 3):
        fig.update_xaxes(title_text='Predicted', row=i, col=j)
        fig.update_yaxes(title_text='True', autorange='reversed', row=i, col=j)

fig.show()
save_figure(fig)
```

```
Warning: Could not save figure (kaleido required):
Image export using the "kaleido" engine requires the kaleido package,
which can be installed using pip:
    $ pip install -U kaleido
```

In [18]:

```python
# Per-Class F1 Scores with Plotly
fig = go.Figure()

for model_name, metrics in per_class_metrics.items():
    color = CONFIG['colors'][model_name]
    fig.add_trace(go.Bar(
        x=list(drone_classes), y=metrics['f1'],
        name=model_name, marker_color=color, opacity=0.8
    ))

fig.update_layout(
    title='Per-Class F1-Score Comparison',
    xaxis_title='Class',
    yaxis_title='F1-Score',
    yaxis_range=[0, 1.1],
    barmode='group',
    height=600, width=1000
)
```

```
fig.show()
save_figure(fig)
```

```
Warning: Could not save figure (kaleido required):
Image export using the "kaleido" engine requires the kaleido package,
which can be installed using pip:
    $ pip install -U kaleido
```

In [19]:

```python
# Inference Time Comparison with Plotly
fig = go.Figure()

models = results_df['Model'].values
p50_times = results_df['Inference_p50_ms'].values
p95_times = results_df['Inference_p95_ms'].values
colors_list = [CONFIG['colors'][m] for m in models]

fig.add_trace(go.Bar(
    y=list(models), x=p50_times, orientation='h',
    marker_color=colors_list, name='p50 (median)',
    text=[f"p50: {t:.2f}ms" for t in p50_times], textposition='outside'
))

# Add p95 as error bars
fig.add_trace(go.Scatter(
    y=list(models), x=p95_times, mode='markers',
    marker=dict(symbol='line-ns', size=15, color='gray', line_width=2),
    name='p95'
))

fig.update_layout(
    title='Model Inference Time Comparison - Single Sample (p50 with p95 markers)',
    xaxis_title='Inference Time per Sample (ms)',
    height=500, width=900,
    showlegend=True
)

fig.show()
save_figure(fig)
```

```
Warning: Could not save figure (kaleido required):
Image export using the "kaleido" engine requires the kaleido package,
which can be installed using pip:
    $ pip install -U kaleido
```

In [20]:
```python
# Model Size Comparison with Plotly
fig = go.Figure()

models = results_df['Model'].values
sizes = results_df['Model_Size_MB'].values
```

```
colors_list = [CONFIG['colors'][m] for m in models]

fig.add_trace(go.Bar(
    y=list(models), x=sizes, orientation='h',
    marker_color=colors_list,
    text=[f"{s:.2f} MB" for s in sizes], textposition='outside'
))

fig.update_layout(
    title='Model Size Comparison',
    xaxis_title='Model Size (MB)',
    height=500, width=900
)

fig.show()
save_figure(fig)
```

```
Warning: Could not save figure (kaleido required):
Image export using the "kaleido" engine requires the kaleido package,
which can be installed using pip:
    $ pip install -U kaleido
```

In [21]:
```python
# Radar Chart with Plotly
def normalize(values):
    min_val = np.min(values)
    max_val = np.max(values)
    if max_val == min_val:
```

```
        return np.ones_like(values)
    return (values - min_val) / (max_val - min_val)

accuracy_norm = results_df['Accuracy'].values
f1_norm = results_df['F1-Score'].values
time_norm = 1 - normalize(results_df['Inference_p50_ms'].values)
size_norm = 1 - normalize(results_df['Model_Size_MB'].values)

categories = ['Accuracy', 'F1-Score', 'Speed (inverse time)', 'Compactness (inverse size)']

fig = go.Figure()

for idx, model in enumerate(models):
    values = [accuracy_norm[idx], f1_norm[idx], time_norm[idx], size_norm[idx]]
    # Close the polygon
    values.append(values[0])
    categories_closed = categories + [categories[0]]

    color = CONFIG['colors'][model]
    fig.add_trace(go.Scatterpolar(
        r=values, theta=categories_closed,
        fill='toself', name=model,
        line_color=color, fillcolor=color, opacity=0.3
    ))

fig.update_layout(
    title='Multi-Metric Model Comparison (Normalized)',
    polar=dict(radialaxis=dict(visible=True, range=[0, 1])),
    height=700, width=800,
    showlegend=True
)

fig.show()
save_figure(fig)
```

```
Warning: Could not save figure (kaleido required):
Image export using the "kaleido" engine requires the kaleido package,
which can be installed using pip:
    $ pip install -U kaleido
```

12. Statistical Tests

In [22]:

```python
# McNemar's Test
def mcnemar_test(y_true, pred1, pred2):
    """McNemar's test for paired predictions."""
    correct1 = (pred1 == y_true)
    correct2 = (pred2 == y_true)

    n01 = np.sum(~correct1 & correct2)
    n10 = np.sum(correct1 & ~correct2)

    if n01 + n10 == 0:
        return 1.0, 0.0

    statistic = (abs(n01 - n10) - 1)**2 / (n01 + n10)
    p_value = 1 - chi2.cdf(statistic, df=1)

    return p_value, statistic

model_names = ['SVM', 'VGG16', 'ResNet50', 'RFUAVNet']
mcnemar_results = []

print("=== McNemar's Test (Pairwise) ===")
print(f"{'Model 1':<12} {'Model 2':<12} {'p-value':<12} {'Statistic':<12} {'Significant'}")
print("-" * 65)

for i in range(len(model_names)):
    for j in range(i+1, len(model_names)):
        model1 = model_names[i]
        model2 = model_names[j]

        p_value, stat = mcnemar_test(
            y_test_psd,
```

```
            predictions[model1],
            predictions[model2]
        )

        significant = "Yes (p<0.05)" if p_value < 0.05 else "No"
        print(f"{model1:<12} {model2:<12} {p_value:<12.4f} {stat:<12.4f} {significant}")

        mcnemar_results.append({
            'Model1': model1,
            'Model2': model2,
            'p_value': p_value,
            'statistic': stat,
            'significant': p_value < 0.05
        })
```

```
=== McNemar's Test (Pairwise) ===
Model 1      Model 2      p-value      Statistic    Significant
--------------------------------------------------------------------
SVM          VGG16        0.0000       28.9515      Yes (p<0.05)
SVM          ResNet50     0.0000       115.9002     Yes (p<0.05)
SVM          RFUAVNet     0.0000       885.2849     Yes (p<0.05)
VGG16        ResNet50     0.0000       38.3432      Yes (p<0.05)
VGG16        RFUAVNet     0.0000       615.0130     Yes (p<0.05)
ResNet50     RFUAVNet     0.0000       398.8173     Yes (p<0.05)
```

In [23]:

```
# Bootstrap Confidence Intervals
def bootstrap_ci(y_true, y_pred, n_iterations=1000, ci=0.95, random_state=42):
    """Bootstrap confidence interval for accuracy."""
    np.random.seed(random_state)
    n_samples = len(y_true)
    accuracies = []

    for _ in range(n_iterations):
        indices = np.random.choice(n_samples, size=n_samples, replace=True)
        y_true_boot = y_true[indices]
        y_pred_boot = y_pred[indices]
        accuracies.append(accuracy_score(y_true_boot, y_pred_boot))

    alpha = 1 - ci
    lower = np.percentile(accuracies, 100 * alpha / 2)
    upper = np.percentile(accuracies, 100 * (1 - alpha / 2))
```

```
    return lower, upper

print("\n=== Bootstrap 95% Confidence Intervals (Accuracy) ===")
print(f"{'Model':<12} {'Accuracy':<12} {'CI Lower':<12} {'CI Upper':<12} {'CI Width'}")
print("-" * 60)

for model_name in model_names:
    acc = accuracy_score(y_test_psd, predictions[model_name])
    lower, upper = bootstrap_ci(
        y_test_psd,
        predictions[model_name],
        n_iterations=CONFIG['n_bootstrap'],
        random_state=CONFIG['random_state']
    )
    width = upper - lower
    print(f"{model_name:<12} {acc:<12.4f} {lower:<12.4f} {upper:<12.4f} {width:.4f}")
```

```
=== Bootstrap 95% Confidence Intervals (Accuracy) ===
Model        Accuracy     CI Lower     CI Upper     CI Width
------------------------------------------------------------
SVM          0.8293       0.8168       0.8404       0.0237
VGG16        0.7851       0.7731       0.7980       0.0249
ResNet50     0.7368       0.7229       0.7520       0.0291
RFUAVNet     0.5289       0.5130       0.5444       0.0314
```

In [24]:

```
# Cohen's Kappa
print("\n=== Cohen's Kappa (Agreement beyond chance) ===")
print(f"{'Model':<12} {'Kappa':<12} {'Interpretation'}")
print("-" * 50)

def interpret_kappa(kappa):
    if kappa < 0:
        return "Poor (worse than random)"
    elif kappa < 0.20:
        return "Slight"
    elif kappa < 0.40:
        return "Fair"
    elif kappa < 0.60:
        return "Moderate"
    elif kappa < 0.80:
```

```
        return "Substantial"
    else:
        return "Almost perfect"

for model_name in model_names:
    kappa = cohen_kappa_score(y_test_psd, predictions[model_name])
    interpretation = interpret_kappa(kappa)
    print(f"{model_name:<12} {kappa:<12.4f} {interpretation}")
```

```
=== Cohen's Kappa (Agreement beyond chance) ===
Model        Kappa        Interpretation
--------------------------------------------------
SVM          0.7998       Substantial
VGG16        0.7484       Substantial
ResNet50     0.6919       Substantial
RFUAVNet     0.4487       Moderate
```

13. Error Analysis

In [25]:

```
# Misclassification Patterns
best_model = results_df.iloc[0]['Model']
best_preds = predictions[best_model]

misclassified_mask = (best_preds != y_test_psd)
misclassified_true = y_test_psd[misclassified_mask]
misclassified_pred = best_preds[misclassified_mask]

print(f"\n=== {best_model} Error Analysis ===")
print(f"Total test samples: {len(y_test_psd)}")
print(f"Misclassified samples: {np.sum(misclassified_mask)} ({100*np.sum(misclassified_mask)/len(y_test_psd):.2f}%)")

print("\nMost common misclassification pairs (True -> Predicted):")
misclass_pairs = list(zip(misclassified_true, misclassified_pred))
unique_pairs, counts = np.unique(misclass_pairs, axis=0, return_counts=True)
sorted_indices = np.argsort(-counts)[:10]

print("  TRUE -> PRED")
for idx in sorted_indices:
    true_cls = drone_classes[unique_pairs[idx][0]]
    pred_cls = drone_classes[unique_pairs[idx][1]]
```

```
    count = counts[idx]
    pct = 100 * count / np.sum(misclassified_mask)
    print(f"  {true_cls} -> {pred_cls}: {count} errors ({pct:.1f}% of errors)")
```

```
=== SVM Error Analysis ===
Total test samples: 3891
Misclassified samples: 664 (17.07%)

Most common misclassification pairs (True -> Predicted):
  TRUE -> PRED
  MA1 -> AIR: 115 errors (17.3% of errors)
  MAV -> MA1: 105 errors (15.8% of errors)
  MA1 -> MAV: 97 errors (14.6% of errors)
  MAV -> AIR: 96 errors (14.5% of errors)
  AIR -> MA1: 85 errors (12.8% of errors)
  AIR -> MAV: 28 errors (4.2% of errors)
  AIR -> INS: 25 errors (3.8% of errors)
  MA1 -> INS: 19 errors (2.9% of errors)
  MAV -> INS: 16 errors (2.4% of errors)
  PHA -> INS: 12 errors (1.8% of errors)
```

In [26]:
```
# Model Agreement Analysis
pred_matrix = np.array([predictions[m] for m in model_names]).T

all_agree = np.all(pred_matrix == pred_matrix[:, 0:1], axis=1)
all_correct = np.all(pred_matrix == y_test_psd[:, None], axis=1)
all_wrong = np.all(pred_matrix != y_test_psd[:, None], axis=1)

print("\n=== Model Agreement Analysis ===")
print(f"Samples where all models agree: {np.sum(all_agree)} ({100*np.sum(all_agree)/len(y_test_psd):.2f}%)")
print(f"Samples where all models are correct: {np.sum(all_correct)} ({100*np.sum(all_correct)/len(y_test_psd):.2f}%)")
print(f"Samples where all models are wrong: {np.sum(all_wrong)} ({100*np.sum(all_wrong)/len(y_test_psd):.2f}%)")

no_model_correct = np.all(pred_matrix != y_test_psd[:, None], axis=1)
print(f"\nHard samples (no model correct): {np.sum(no_model_correct)}")
if np.sum(no_model_correct) > 0:
    hard_true = y_test_psd[no_model_correct]
    unique_hard, counts_hard = np.unique(hard_true, return_counts=True)
    print("Distribution by class:")
    for cls_idx, count in zip(unique_hard, counts_hard):
        print(f"  {drone_classes[cls_idx]}: {count} samples")
```

```
=== Model Agreement Analysis ===
Samples where all models agree: 1411 (36.26%)
Samples where all models are correct: 1408 (36.19%)
Samples where all models are wrong: 126 (3.24%)

Hard samples (no model correct): 126
Distribution by class:
  AIR: 17 samples
  DIS: 6 samples
  MA1: 45 samples
  MAV: 57 samples
  PHA: 1 samples
```

14. Summary

In [27]:

```python
print("=" * 60)
print("MODEL COMPARISON SUMMARY")
print("=" * 60)

print("\nRANKING (by Accuracy):")
for idx, row in results_df.iterrows():
    print(f"  {idx+1}. {row['Model']:<10} Acc={row['Accuracy']:.4f} F1={row['F1-Score']:.4f}")

best = results_df.iloc[0]
fastest = results_df.loc[results_df['Inference_p50_ms'].idxmin()]
smallest = results_df.loc[results_df['Model_Size_MB'].idxmin()]

print(f"\nBest accuracy: {best['Model']} ({best['Accuracy']:.4f})")
print(f"Fastest inference: {fastest['Model']} (p50={fastest['Inference_p50_ms']:.2f}ms)")
print(f"Smallest model: {smallest['Model']} ({smallest['Model_Size_MB']:.2f}MB)")

print("=" * 60)
```

```
============================================================
MODEL COMPARISON SUMMARY
============================================================

RANKING (by Accuracy):
  1. SVM        Acc=0.8293 F1=0.8277
  2. VGG16      Acc=0.7851 F1=0.7839
  3. ResNet50   Acc=0.7368 F1=0.7315
  4. RFUAVNet   Acc=0.5289 F1=0.5239

Best accuracy: SVM (0.8293)
Fastest inference: RFUAVNet (p50=1.49ms)
Smallest model: RFUAVNet (0.06MB)
============================================================
```

15. Export Results

In [28]:

```python
# Save results DataFrame
results_path = os.path.join(CONFIG['models_dir'], 'model_comparison_results.csv')
results_df.to_csv(results_path, index=False)
print(f"Results saved to {results_path}")

# Save detailed comparison
comparison_dict = {
    'results_df': results_df,
    'per_class_metrics': per_class_metrics,
    'mcnemar_results': mcnemar_results,
    'predictions': predictions,
    'true_labels': y_test_psd,
    'drone_classes': drone_classes,
    'config': CONFIG
}

comparison_path = os.path.join(CONFIG['models_dir'], 'model_comparison_full.pkl')
with open(comparison_path, 'wb') as f:
    pickle.dump(comparison_dict, f)
print(f"Full comparison saved to {comparison_path}")

# Save export summary
```

```python
if 'export_df' in globals():
    export_summary_path = os.path.join(CONFIG['sample_export_dir'], 'export_summary.csv')
    export_df.to_csv(export_summary_path, index=False)
    print(f"Export summary saved to {export_summary_path}")
```

```
Results saved to drive/MyDrive/DroneDetect_V2/output/models/model_comparison_results.csv
Full comparison saved to drive/MyDrive/DroneDetect_V2/output/models/model_comparison_full.pkl
```