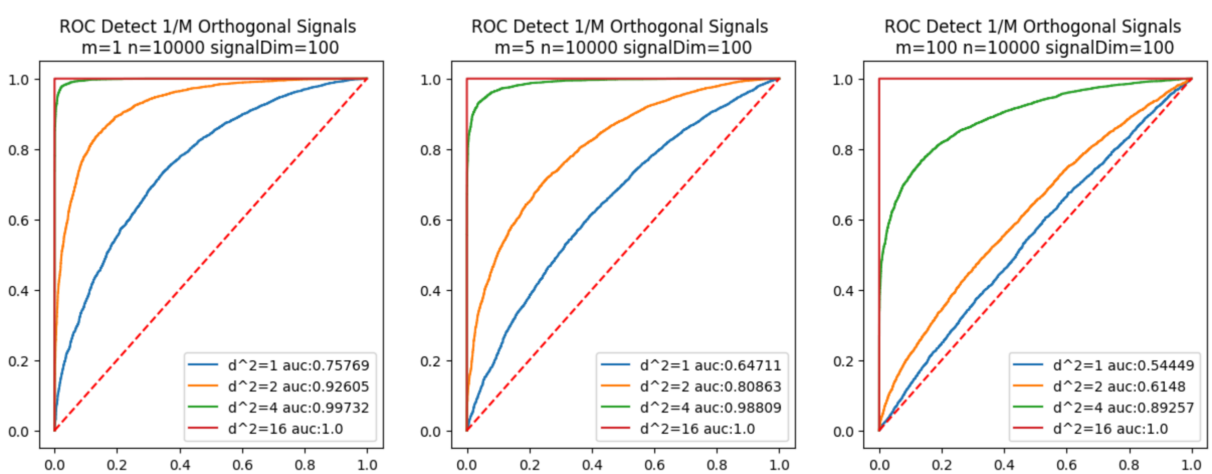
1a) 1/M orthogonal signals (any exists)

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics

# M = num signals to generate
# N = dimension of signals
def generate_orthogonal_signals(M, N):
    signals = np.random.normal(size=(M, N))
    orthogonal_signals = np.zeros((M, N))

    for i in range(M):
        orthogonal_signals[i] = signals[i]
        for j in range(i):
            orthogonal_signals[i] -= np.dot(orthogonal_signals[j], signals[i]) / np

        # Normalize the signal
        orthogonal_signals[i] /= np.linalg.norm(orthogonal_signals[i])

    priors = np.random.dirichlet(np.ones(M),size=1)

    return orthogonal_signals, priors

def calcPdPfa(data):
    h1s = data[data[:,1] == 1].shape[0]
    h0s = data[data[:,1] == 0].shape[0]
    currh1 = h1s
    currh0 = h0s


    pd = [1]
    pfa = [1]
    if h0s == 0:
        pfa.append(0)
        pd.append(1)
    if h1s == 0:
        pfa.append(1)
        pd.append(0)
    for i in data:
        if i[1] == 1:
            currh1 -= 1
        elif i[1] == 0:
            currh0 -= 1
        pfa.append(0 if h0s == 0 else currh0/h0s)
        pd.append(0 if h1s == 0 else currh1/h1s)
    pd.append(0)
    pfa.append(0)
    return pd,pfa
```

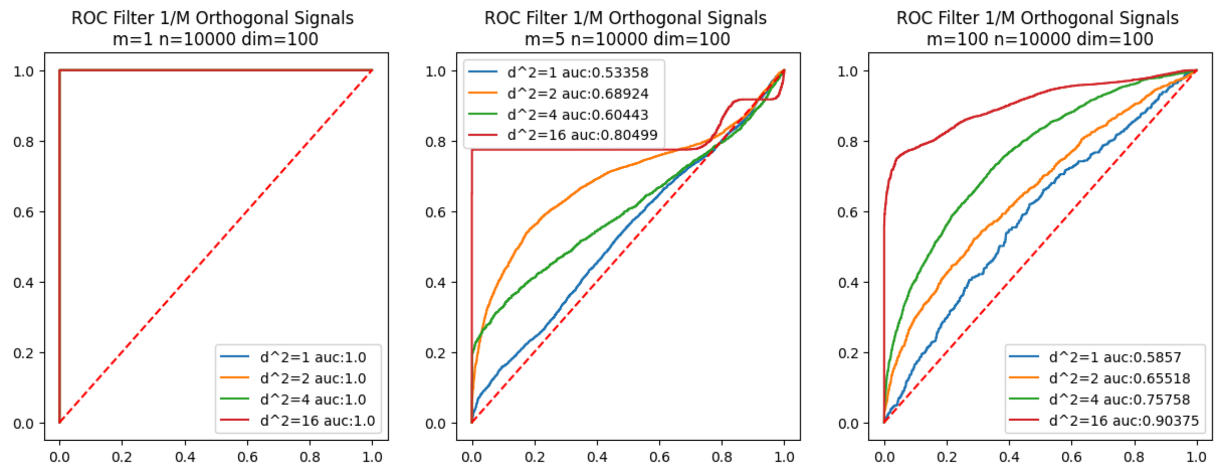
In [ ]:
```
np.random.seed(seed=0)
M = [1,5,100]
```

```
d2 = [1,2,4,16]
E = 1
signalDim = 100
n = 10000

def detect_1_M(x, noise_variance, m_orthogonal_signals):
    prodSum = E/noise_variance * (x @ m_orthogonal_signals.T)
    # print("prodsum",prodSum)
    exp = np.exp(prodSum)
    # print("exp",exp)
    lambdaa = (1/m) * np.sum(exp, axis=1)
    # print("lambda",lambdaa)
    return lambdaa
fig, ax = plt.subplots(1,len(M), figsize=(15,5))
for i,m in enumerate(M):
    for d in d2:
        noise_variance = E/d
        m_orthogonal_signals, _ = generate_orthogonal_signals(m, signalDim)
        rand_m = np.random.choice(m, n//2)


        h1 = m_orthogonal_signals[rand_m] + np.random.normal(0, noise_variance, siz
        h0 = np.random.normal(0, noise_variance, size=(n//2, signalDim))
        decisionStats = np.ndarray((n,2))

        decisionStats[n//2:,0] = detect_1_M(h0, noise_variance, m_orthogonal_signal
        decisionStats[n//2:,1] = 0
        decisionStats[:n//2,0] = detect_1_M(h1, noise_variance, m_orthogonal_signal
        decisionStats[:n//2,1] = 1

        decisionStats = decisionStats[decisionStats[:, 0].argsort()]
        pd,pfa = calcPdPfa(decisionStats)
        auc = metrics.auc(pfa,pd)
        ax[i].plot(pfa,pd, label=f"d^2={d} auc:{round(auc,5)}")
        ax[i].legend()
        ax[i].set_title(f"ROC Detect 1/M Orthogonal Signals \nm={m} n={n} signalDim
    ax[i].plot([0, 1], [0, 1],'r--')
plt.show()
```



1b) 1/M orthogonal signals (which one exists)

In [ ]:
```
# np.random.seed(seed=0)
M = [1,5,100]
d2 = [1,2,4,16]
E = 1
signalDim = 100
n = 10000

def filter_1_M(x, priors, m_orthogonal_signals):
    # print(x[0])
    # print("ortho",m_orthogonal_signals)
    dotprod = (x @ m_orthogonal_signals.T)
    # print("dotprod",dotprod[0])
    weighted_dotprod = dotprod * priors
    # print(weighted_dotprod)
    result = np.argmax(weighted_dotprod, axis=1)
    # print("res", result)
    lambdaa = weighted_dotprod[np.arange(weighted_dotprod.shape[0]), result]
    # print("lambda",lambdaa)
    return lambdaa,result

fig, ax = plt.subplots(1,len(M), figsize=(15,5))
for i,m in enumerate(M):
    for d in d2:
        noise_variance = E/d
        m_orthogonal_signals, priors = generate_orthogonal_signals(m, signalDim)
        rand_m = np.random.choice(m, n)

        signals = m_orthogonal_signals[rand_m] + np.random.normal(0, noise_variance
        decisionStats = np.ndarray((n,2))

        lambdaa,result = filter_1_M(signals, priors, m_orthogonal_signals)
        # print(rand_m)
        # print(np.where(rand_m==result,1,0))

        decisionStats[:,0] = lambdaa
        decisionStats[:,1] = np.where(rand_m==result,1,0)

        decisionStats = decisionStats[decisionStats[:, 0].argsort()]

        pd,pfa = calcPdPfa(decisionStats)
        auc = metrics.auc(pfa,pd)
        ax[i].plot(pfa,pd, label=f"d^2={d} auc:{round(auc,5)}")
        ax[i].legend()
        ax[i].set_title(f"ROC Filter 1/M Orthogonal Signals \nm={m} n={n} dim={sign

    ax[i].plot([0, 1], [0, 1],'r--')
plt.show()
```

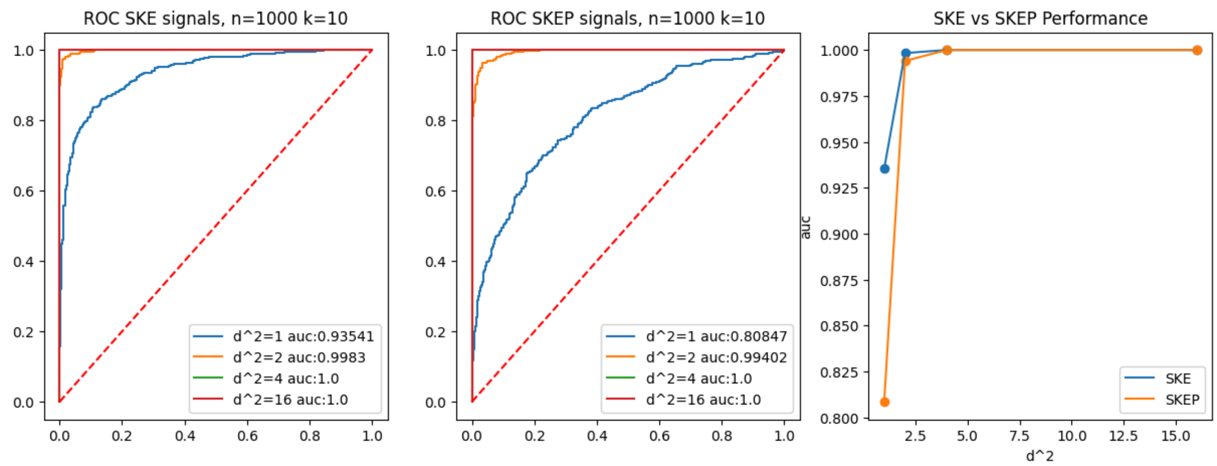


2) SKEP

In [ ]:
```
np.random.seed(seed=0)

D2 = [1,2,4,16]
E = 1
FREQ = 10
K=10
N = 1000
T = np.linspace(0, 2 * np.pi, K)

def simulate(signal, ax, title, filter):
    ret = []
    for d in D2:
        noise_variance = E/d

        h1 = signal + np.random.normal(0, noise_variance, size=(N//2, K))
        h0 = np.random.normal(0, noise_variance, size=(N//2, K))

        decisionStats = filter(h1, h0)
        decisionStats = decisionStats[decisionStats[:, 0].argsort()]

        pd,pfa = calcPdPfa(decisionStats)
        auc = metrics.auc(pfa,pd)
        ax.plot(pfa,pd, label=f"d^2={d} auc:{round(auc,5)}")
        ret.append(auc)

    ax.legend()
    ax.set_title(title)
    ax.plot([0, 1], [0, 1],'r--')
    return ret

def ske(signal, ax, title):
    def filter(h1, h0):
        decisionStats = np.ndarray((N,2))
        decisionStats[:N//2,0] = h1 @ signal.T
        decisionStats[:N//2,1] = 1

        decisionStats[N//2:,0] = h0 @ signal.T
```

```
        decisionStats[N//2:,1] = 0
        return decisionStats
    return simulate(signal,ax,title,filter)

def skep(signal, amp, ax, title):
    def filter(h1, h0):
        decisionStats = np.ndarray((N,2))
        decisionStats[:N//2,0] = np.power(np.sum(h1*amp*np.cos(FREQ * T), axis=1),2
        decisionStats[:N//2,1] = 1

        decisionStats[N//2:,0] = np.power(np.sum(h0*amp*np.cos(FREQ * T), axis=1),2
        decisionStats[N//2:,1] = 0
        return decisionStats
    return simulate(signal,ax,title,filter)

def skea(signal, phase, ax, title):
    def filter(h1, h0):
        decisionStats = np.ndarray((N,2))
        decisionStats[:N//2,0] = np.sum(h1*np.sin(FREQ * T + phase), axis=1)
        decisionStats[:N//2,1] = 1

        decisionStats[N//2:,0] = np.sum(h0*np.sin(FREQ * T + phase), axis=1)
        decisionStats[N//2:,1] = 0
        return decisionStats
    return simulate(signal,ax,title,filter)

def compareROC(data,ax, title):
    for label, d in data.items():
        ax.plot(D2, d, label=label)
        ax.scatter(D2, d)
    ax.set_xlabel("d^2")
    ax.set_ylabel("auc")
    ax.set_title(title)
    ax.legend()
```

In [ ]:
```
np.random.seed(seed=0)

unknownPhase = np.random.uniform(0, 2*np.pi)
amp = 1
signal = amp * np.sin(FREQ * T + unknownPhase)

fig, ax = plt.subplots(1,3,figsize=(15,5))
data = {}
data['SKE'] = ske(signal,ax[0], f"ROC SKE signals, n={N} k={K}")
data['SKEP'] = skep(signal, amp, ax[1], f"ROC SKEP signals, n={N} k={K}")

compareROC(data, ax[2],"SKE vs SKEP Performance")
plt.show()
```

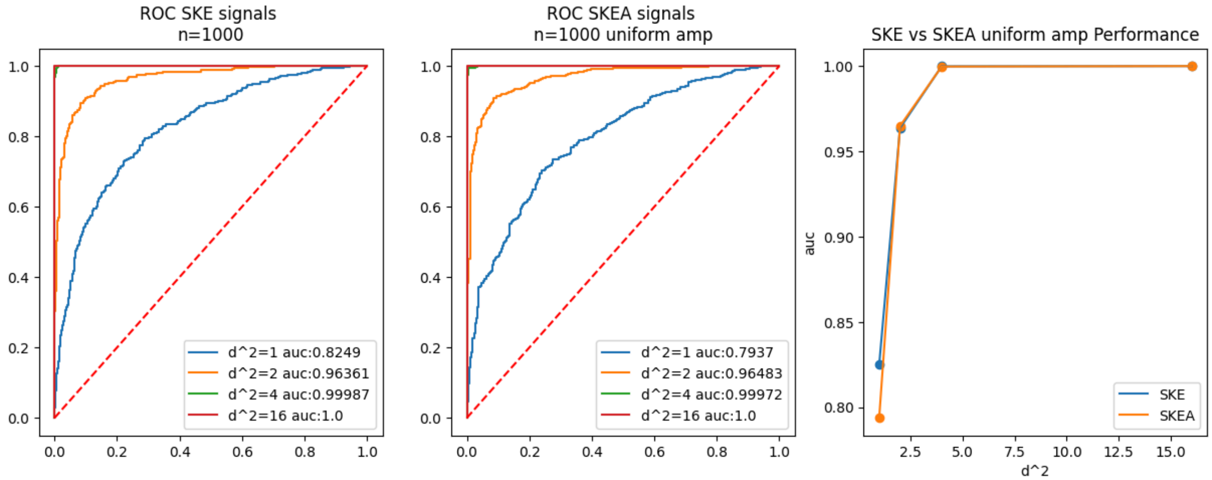


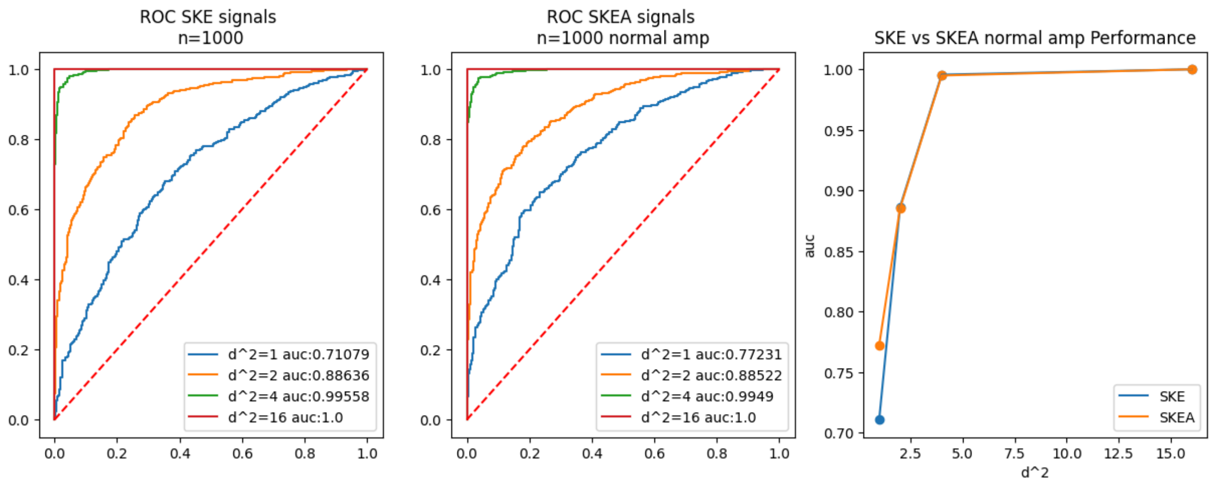
The ROC for SKEP and SKE show that performance decreases when there are more unknowns, which is expected. As d^2 decreases, noise variance increases, so the performance of SKE and SKEP decrease. However, SKEP performance gets worse faster than SKE.

3) SKEA

In [ ]:

```
np.random.seed(seed=0)

# uniform amp
unknownAmp = np.random.uniform(0,1)
phase = np.random.uniform(0, 2*np.pi)
signal = unknownAmp * np.sin(FREQ * T + phase)
fig, ax = plt.subplots(1,3,figsize=(15,5))
data = {}
data['SKE'] = ske(signal, ax[0], f"ROC SKE signals \nn={n}")
data['SKEA'] = skea(signal, phase, ax[1], f"ROC SKEA signals \nn={n} uniform amp")
compareROC(data, ax[2],"SKE vs SKEA uniform amp Performance")
plt.show()

# normal amp
unknownAmp = abs(np.random.normal(0,1))
phase = np.random.uniform(0, 2*np.pi)
signal = unknownAmp * np.sin(FREQ * T + phase)
fig, ax = plt.subplots(1,3,figsize=(15,5))
data = {}
data['SKE'] = ske(signal, ax[0], f"ROC SKE signals \nn={n}")
data['SKEA'] = skea(signal, phase, ax[1],f"ROC SKEA signals \nn={n} normal amp")
compareROC(data, ax[2],"SKE vs SKEA normal amp Performance")

plt.show()
```





Like SKEP, the ROC for SKEA and SKE show that performance decreases when there are more unknowns, which is expected. As d^2 decreases, noise variance increases, so the performance of SKE and SKEA decrease. When we assume that the unknown amplitude is drawn from a uniform[0,1] distribution, performance is only worse than the SKE case when $d^2 = 1$. When we assume that the unknown amplitude is drawn from a Normal[0,1] distribution, performance only really differs from the SKE case when $d^2 = 1$, but this time the performance is better, with AUC for the SKE case being 0.71 compared to the AUC for the SKEA case being 0.77.

All together

In [ ]:
```
np.random.seed(seed=0)

amp = np.random.uniform(0.0,1.0)
phase = np.random.uniform(0, 2*np.pi)
signal = amp * np.sin(FREQ * T + phase)

fig, ax = plt.subplots(1,3,figsize=(15,5))
data = {}
data['SKE'] = ske(signal, ax[0], f"ROC SKE signals \nn={N}, k={K}")
data['SKEA'] = skea(signal, phase, ax[1],f"ROC SKEA signals \nn={N}, k={K}")
```

```
data['SKEP'] = skep(signal, amp, ax[2],f"ROC SKEP signals \nn={N}, k={K}")
plt.show()

fig, ax = plt.subplots()
compareROC(data, ax,"SKE vs SKEP vs SKEA Performance")
plt.show()
```