1.

```python
import pandas as pd
import numpy as np

def transform_points(data):
  x1 = data[:, 0]
  x2 = data[:, 1]
  transformed = np.column_stack((
      np.ones(len(x1)),
      x1,
      x2,
      x1**2,
      x2**2,
      x1 * x2,
      np.abs(x1 - x2),
      np.abs(x1 + x2)
  ))
  return transformed

def model3(data):
  x1 = data[:, 0]
  x2 = data[:, 1]
  transformed = np.column_stack((
      np.ones(len(x1)),
      x1,
      x2,
      x1**2
  ))
  return transformed
```

```python
def model4(data):
  x1 = data[:, 0]
  x2 = data[:, 1]
  transformed = np.column_stack((
      np.ones(len(x1)),
      x1,
      x2,
      x1**2,
      x2**2
  ))
  return transformed

def model5(data):
  x1 = data[:, 0]
  x2 = data[:, 1]
  transformed = np.column_stack((
      np.ones(len(x1)),
      x1,
      x2,
      x1**2,
      x2**2,
      x1 * x2
  ))
  return transformed
```

```python
def model6(data):
  x1 = data[:, 0]
  x2 = data[:, 1]
  transformed = np.column_stack((
      np.ones(len(x1)),
      x1,
      x2,
      x1**2,
      x2**2,
      x1 * x2,
      np.abs(x1 - x2)
  ))
  return transformed

def linear_regression(data, Y):
  transpose = data.T
  inv = np.linalg.inv(np.dot(transpose, data))
  w = np.dot(np.dot(inv, transpose), Y)
  return w

def find_error(X, Y, w):
  Y_pred = np.sign(np.dot(X, w))
  return np.sum(Y_pred != Y) / len(Y)

train = np.loadtxt('in.dta.txt')[:25]
validation = np.loadtxt('in.dta.txt')[25:]
```

```python
weights3 = linear_regression(model3(train), train[:, 2])
weights4 = linear_regression(model4(train), train[:, 2])
weights5 = linear_regression(model5(train), train[:, 2])
weights6 = linear_regression(model6(train), train[:, 2])
weights = linear_regression(transform_points(train), train[:, 2])

model3_validation_error = find_error(model3(validation), validation[:, 2], weights3)
model4_validation_error = find_error(model4(validation), validation[:, 2], weights4)
model5_validation_error = find_error(model5(validation), validation[:, 2], weights5)
model6_validation_error = find_error(model6(validation), validation[:, 2], weights6)
validation_error = find_error(transform_points(validation), validation[:, 2], weights)


print("k = 3: ", model3_validation_error)
print("k = 4: ", model4_validation_error)
print("k = 5: ", model5_validation_error)
print("k = 6: ", model6_validation_error)
print("k = 7: ", validation_error)
```

```
k = 3:  0.3
k = 4:  0.5
k = 5:  0.2
k = 6:  0.0
k = 7:  0.1
```

Based on my code above, the classification error on the validation set is smallest for k=6 (the error is 0). => [d]

2.

```
test = np.loadtxt('out.dta.txt')

out3_error = find_error(model3(test), test[:, 2], weights3)
out4_error = find_error(model4(test), test[:, 2], weights4)
out5_error = find_error(model5(test), test[:, 2], weights5)
out6_error = find_error(model6(test), test[:, 2], weights6)
out_error = find_error(transform_points(test), test[:, 2], weights)


print("k = 3: ", out3_error)
print("k = 4: ", out4_error)
print("k = 5: ", out5_error)
print("k = 6: ", out6_error)
print("k = 7: ", out_error)
```

```
k = 3:  0.42
k = 4:  0.416
k = 5:  0.188
k = 6:  0.084
k = 7:  0.072
```

Based on my code above, the out-of-sample error is smallest for $k=7$, where it is 0.072.

=> [e]

3.

```
train = np.loadtxt('in.dta.txt')[25:]
validation = np.loadtxt('in.dta.txt')[:25]

weights3 = linear_regression(model3(train), train[:, 2])
weights4 = linear_regression(model4(train), train[:, 2])
weights5 = linear_regression(model5(train), train[:, 2])
weights6 = linear_regression(model6(train), train[:, 2])
weights = linear_regression(transform_points(train), train[:, 2])

model3_validation_error = find_error(model3(validation), validation[:, 2], weights3)
model4_validation_error = find_error(model4(validation), validation[:, 2], weights4)
model5_validation_error = find_error(model5(validation), validation[:, 2], weights5)
model6_validation_error = find_error(model6(validation), validation[:, 2], weights6)
validation_error = find_error(transform_points(validation), validation[:, 2], weights)


print("k = 3: ", model3_validation_error)
print("k = 4: ", model4_validation_error)
print("k = 5: ", model5_validation_error)
print("k = 6: ", model6_validation_error)
print("k = 7: ", validation_error)
```

```
k = 3:  0.28
k = 4:  0.36
k = 5:  0.2
k = 6:  0.08
k = 7:  0.12
```

Based on my code above, when the amount of training and test samples are switched, the error on validation set is smallest for $k=6$ at 0.08 => [d]

4.

```
out3_error = find_error(model3(test), test[:, 2], weights3)
out4_error = find_error(model4(test), test[:, 2], weights4)
out5_error = find_error(model5(test), test[:, 2], weights5)
out6_error = find_error(model6(test), test[:, 2], weights6)
out_error = find_error(transform_points(test), test[:, 2], weights)


print("k = 3: ", out3_error)
print("k = 4: ", out4_error)
print("k = 5: ", out5_error)
print("k = 6: ", out6_error)
print("k = 7: ", out_error)
```

```
k = 3:  0.396
k = 4:  0.388
k = 5:  0.284
k = 6:  0.192
k = 7:  0.196
```

Based on my code above, the out-of-sample error is smallest for $k=6$ (at 0.192) => [d]

5.

```
#problem 1: 0.084, problem 3: 0.192
final = np.array([0.084, 0.192])
option1 = np.array([0, 0.1])
option2 = np.array([0.1, 0.2])
option3 = np.array([0.1, 0.3])
option4 = np.array([0.2, 0.2])
option5 = np.array([0.2, 0.3])
print(np.linalg.norm(final - option1))
print(np.linalg.norm(final - option2))
print(np.linalg.norm(final - option3))
print(np.linalg.norm(final - option4))
print(np.linalg.norm(final - option5))
```

```
0.12457929201917949
0.01788854381999832
0.10917875251164944
0.11627553482998906
0.15849290204927158
```

The out-of-sample values of 0.084 and 0.192 from the models from problems 1 and 3 are closest to 0.1 and 0.2, as shown in my code above. => [b]

6. e_1 and e_2 are distributed uniformly on $[0,1]$ $e = \min(e_1, e_2)$

Since e_1 and e_2 are distributed uniformly on $[0,1]$, their expected values are each 0.5, so all answer choices are still valid.

From lecture, we know that:

$e > \theta$ iff $e_1 > \theta$ and $e_2 > \theta$ and e_1 is independent of e_2 $\quad P(e > \theta) = (1-\theta)(1-\theta)$

$P(e_1 > \theta)$ $P(e_2 > \theta)$

we want to find the average value of $(1-\theta)(1-\theta)$ on $[0,1]$, so we can take $\frac{1}{N}\int_0^N f(\theta)\,d\theta$ where $N = 1$:

$$\int_0^1 (1-\theta)(1-\theta)\,d\theta = \int_0^1 (1-2\theta+\theta^2)\,d\theta = \left[\theta - \theta^2 + \frac{\theta^3}{3}\right]_0^1$$

$$= (1 - 1 + \tfrac{1}{3}) - (0 - 0 + 0) = \frac{1}{3} \approx 0.33 \Rightarrow \text{this is}$$

closest to 0.4 $\Rightarrow$ [d]

7. (x, y): $(-1, 0), (p, 1), (1, 0)$

$h_0(x) = b$ or $h_1(x) = ax + b$

→ both models use leave-one-out cross validation w/ squared error measure → $N-1$ points for training, 1 for validation

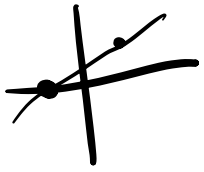
$$\text{error} = \frac{1}{N}\sum_{n=1}^{N} e_n$$

$e_n = E_{val}(g_n^-) = e(g_n^-(x_n), y_n)$ → calculate error for each point as cross-validifying point

use $(-1, 0)$ and $(p, 1)$ for training:

$h_0(x) = 0.5$ so error for $(1, 0) =$ **0.5**

$h_1(x) \Rightarrow$ eq. of line that passes through $(-1, 0)$ and $(p, 1)$



$$\text{slope} = \frac{1}{p+1}$$

$$0 = -1\left(\frac{1}{p+1}\right) + b$$

$$\frac{1}{p+1} = b$$

$$h_1(x) = \frac{1}{p+1}x + \frac{1}{p+1} = \frac{(x+1)}{p+1}$$

error for $(1, 0) =$ $\frac{2}{p+1}$

use $(1, 0)$ and $(p, 1)$ for training:

$h_0(x) = 0.5 \Rightarrow$ error for $(-1, 0) =$ **0.5**

$h_1(x) \Rightarrow$ line passing through: slope $= \frac{1}{p-1}$

$0 = \frac{1}{p-1}(1) + b \quad h_1(x) = \frac{x-1}{p-1}$ error for

$(-1,0)$: $\boxed{\frac{-2}{p-1}}$

use $(1,0)$ and $(-1,0)$ for training:

$h_0(x)=0 \Rightarrow$ error for $(p,1) = \textcircled{1}$

$h_1(x) = 0 \Rightarrow$ error for $(p,1) = \textcircled{1}$

MSE for h_0: $\frac{1}{3}\sqrt{\frac{1}{4}+\frac{1}{4}+1} = \frac{\sqrt{6}}{6}$

MSE for h_1: $\frac{1}{3}\sqrt{\frac{4}{(p+1)^2}+\frac{4}{(p-1)^2}+1} = \frac{\sqrt{6}}{6}$

$$\frac{4}{(p+1)^2}+\frac{4}{(p-1)^2}+1 = \frac{3}{2}$$

$$\left(\frac{1}{(p+1)^2}+\frac{1}{(p-1)^2} = \frac{1}{8}\right.$$

$$\frac{p^2+1}{(p+1)^2(p-1)^2} = \frac{1}{16}$$

$$16p^2+16 = (p+1)^2(p-1)^2$$

using calculator, real solutions are: $p = \pm\sqrt{9+4\sqrt{6}}$

since $p>0 \Rightarrow \boxed{[C]\ \sqrt{9+4\sqrt{6}}}$

8.

```python
from sklearn.svm import SVC
import numpy as np
import random


def classify_point(point, m, b):
    x, y = point
    if y > m * x + b:
      return 1
    return -1

def rand_func():
    x1 = random.uniform(-1.0, 1.0)
    x2 = random.uniform(-1.0, 1.0)
    y1 = random.uniform(-1.0, 1.0)
    y2 = random.uniform(-1.0, 1.0)
    m = (y2 - y1) / (x2 - x1)
    b = y1 - m * x1
    return m, b

def create_PLA_points(N):
    m, b = rand_func()
    X = np.random.uniform(-1.0, 1.0, (N, 2))
    Y = np.array([classify_point(point, m, b) for point in X])
    if len(set(Y)) < 2:
        return create_PLA_points(N)
    return X, Y, m, b
```

[59]

```python
def PLA(weights, X, Y):
    iterations = 0
    while iterations < 10000:
        misclassified = []
        for i, point in enumerate(X):
            if np.sign(np.dot(weights, point)) != Y[i]:
                misclassified.append(i)
        if len(misclassified) == 0:
            break
        idx = np.random.choice(misclassified)
        weights += Y[idx] * X[idx]
        iterations += 1
    return weights

def find_error(m, b, X_test, Y_test, w=None, model=None):
    if model:
        Y_pred = model.predict(X_test)
    else:
        X_test_with_bias = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
        Y_pred = np.sign(np.dot(X_test_with_bias, w))
    return np.mean(Y_pred != Y_test)
```

```python
svm_better = 0
for _ in range(1000):
    X, Y, m, b = create_PLA_points(10)
    X_with_bias = np.hstack((np.ones((X.shape[0], 1)), X))
    weights = np.zeros(X_with_bias.shape[1])
    weights = PLA(weights, X_with_bias, Y)
    X_test = np.random.uniform(-1, 1, (1000, 2))
    Y_test = np.array([classify_point(point, m, b) for point in X_test])
    error = find_error(m, b, X_test, Y_test, w=weights)
    clf = SVC(kernel="linear", C=10**10)
    clf.fit(X, Y)
    svm_error = find_error(m, b, X_test, Y_test, model=clf)

    if svm_error < error:
        svm_better += 1

print(svm_better / 1000)
```

```
0.608
```

Based on my code above, for $N=10$, g_{SVM} is better than g_{PLA} in approximating f 54.8% of the time

$\Rightarrow$ closest to **[C] 60%**

9.

```
svm_better2 = 0
total_SV = 0
for _ in range(1000):
    X, Y, m, b = create_PLA_points(100)
    X_with_bias = np.hstack((np.ones((X.shape[0], 1)), X))
    weights = np.zeros(X_with_bias.shape[1])
    weights = PLA(weights, X_with_bias, Y)
    X_test = np.random.uniform(-1, 1, (1000, 2))
    Y_test = np.array([classify_point(point, m, b) for point in X_test])
    error = find_error(m, b, X_test, Y_test, w=weights)
    clf = SVC(kernel="linear", C=10**10)
    clf.fit(X, Y)
    svm_error = find_error(m, b, X_test, Y_test, model=clf)

    if svm_error < error:
        svm_better2 += 1
    total_SV += len(clf.support_vectors_)

print(svm_better2 / 1000)
print(total_SV/1000)
```

```
0.576
2.995
```

Based on my code above, for $N=100$, g_{SVM} is better than g_{PLA} in approximating f 57.6% of the time => closest to [d] 65%

10. Based on my code in (9), the average number of support vectors for $N=100$ is 3

=> [b]