1.

```
import numpy as np
def flip_coins():
    flips = np.random.randint(0, 2, 10)
    return np.mean(flips)

def experiment():
  proportions = np.random.randint(0, 2, (1000, 10)).mean(axis=1)
  first_coin = proportions[0]
  rand_coin = proportions[np.random.randint(0, 1000)]
  min_coin = proportions.min()
  return first_coin, rand_coin, min_coin

results = np.array([experiment() for _ in range(100000)])
firsts = results[:, 0]
randoms = results[:, 1]
mins = results[:, 2]
print(np.mean(mins))
```

```
0.037689
```

According to my code simulating 10,000 experiments of 1,000 coins each being flipped 10 times, the average for v_{min} is closest to 0.01 => [b]

2. Hoeffding's inequality: $\mathbb{P}[|\nu - \mu| > \varepsilon] \leq 2e^{-2\varepsilon^2 N}$

ν => stored in the arrays

μ => actual probabilities

N = sample size (10)

reasonable ε value is 0.2

for a fair coin, $\mu = 0.5$

for v_{min}, v_{rand}, v_1

$v_{min} = 0.0377$

$v_{rand} = 0.500$

$v_1 = 0.500$

$\mathbb{P}[|v_{min} - \mu| > 0.05] \leq 2e^{-2(0.05)^2 \cdot 10} = 0.899$

```
#choose epsilon = 0.2, so lower bound is 0.3 and upper bound is 0.7
count1 = np.sum((firsts < 0.3) | (firsts > 0.7))
count2 = np.sum((randoms < 0.3) | (randoms > 0.7))
count3 = np.sum((mins < 0.3) | (mins > 0.7))

print("P[|v1 - u| > e]: ", count1 / 100000)
print("P[|vrand - u| > e]: ", count2 / 100000)
print("P[|vmin - u| > e]: ", count3 / 100000)
```

```
P[|v1 - u| > e]:  0.10961
P[|vrand - u| > e]:  0.11074
P[|vmin - u| > e]:  1.0
```

Based on my calculation from the code, v_1 and v_{rand} satisfy Hoeffding's inequality, while v_{min} does not satisfy the inequality. This makes sense because it is probable that the average number of heads in 10 flips from a randomly chosen sample from 1000 samples is close to the real value, and the same goes for the first sample of 1000 samples. However, it is probable that the minimum number of heads of 1000 samples deviates from the true value of 0.5, because not all of the runs of 10 flips are close to 0.5. Final answer: [d] C_1 and C_{rand}

3. $P(y|x) = \begin{cases} \lambda, & y = f(x) \quad \mu\lambda \\ 1-\lambda, & y \neq f(x) \quad (1-\mu)(1-\lambda) \end{cases}$

bin model

hypothesis h, error w/ probability μ for approximating f (target function)

↳ probability of error by h for approximating y?

The probability of error by h for approximating y can be split up into 2 cases: 1 case where h correctly approximates f (probability μ) and 1 case where h does not correctly approximate f (probability $1-\mu$). When h correctly approximates f, y is correctly approximated with probability λ, giving $\mu\lambda$ as the total probability. When h does not correctly approximate f, y is correctly approximated with error $1-\lambda$, giving the total probability of $(1-\mu)(1-\lambda)$ for this case. Adding these two cases gives:

$\mu\lambda + (1-\mu)(1-\lambda) \Rightarrow$ [e]

4. The performance of h will be independent from μ for what λ?

$2\mu\lambda + 1 - \lambda - \mu$

↳ probability of h correctly approximating f

$\mu(2\lambda - 1) + 1 - \lambda$ = probability of error by h for approximating y

↳ want this to be 0

so probability does not depend on μ

$2\lambda - 1 = 0$

$\lambda = \frac{1}{2} \Rightarrow [b]$

5. X : [point 1, point 2, ⋮, point n] Y : [±1, ±1, ±1, ⋮] → classification of points

return $w = (X^T X)^{-1} X^T y$ ⇒ compute error

```python
import random
import numpy as np

N = 100

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 classify_point(point, m, b):
    x, y = point
    if y > m * x + b:
      return 1
    if y == m * x + b:
      return 0
    return -1

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

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

f = []
g = []
errors = []
for _ in range(1000):
  [m, b], X, Y = create_points()
  f.append([m, b])
  X_with_bias = np.hstack((np.ones((X.shape[0], 1)), X))
  x_transpose = X_with_bias.T
  inv = np.linalg.inv(np.dot(x_transpose, X_with_bias))
  w = np.dot(np.dot(inv, x_transpose), Y)
  g.append(w)
  errors.append(find_error(X_with_bias, Y, w))

print(sum(errors) / len(errors))
```

```
0.03801000000000004
```

According to my implementation of linear regression shown above, the average E_{in} over the 1000 experiments is closest to 0.01

↳ [c]

6.

```
[4]  def create_test_points(m, b):
       X = np.random.uniform(-1.0, 1.0, (1000, 2))
       Y = np.array([classify_point(point, m, b) for point in X])
       return X, Y

     misclassified = 0
     errors_out = []
     for i in range(1000):
       X_test, y_g = create_test_points(g[i][0], g[i][1])
       y_actual = np.array([classify_point(point, f[i][0], f[i][1]) for point in X_test])
       X_test_with_bias = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
       errors_out.append(find_error(X_test_with_bias, y_actual, g[i]))

     print(sum(errors) / len(errors))
```

```
0.03801000000000004
```

According to my analysis of E_{out} as shown above, the average value of E_{out} across the 1000 experiments is closest to 0.01 → [c]

7. According to my code, the average number of iterations is closest to 1 → [a]

```
[5] def create_PLA_points():
      m, b = rand_func()
      X = []
      Y = []
      X = np.random.uniform(-1.0, 1.0, (10, 2))
      Y = np.array([classify_point(point, m, b) for point in X])
      return X, Y

    def PLA(weights, X, Y):
      iterations = 0
      while iterations < 10000:
        misclassified = []
        for i, point in enumerate(X):
          if np.sign(np.dot(weights[1:], point) + weights[0]) != Y[i]:
            misclassified.append(i)

        if len(misclassified) == 0:
          break

        idx = np.random.choice(misclassified)
        weights[0] += Y[idx]
        weights[1:] += Y[idx] * X[idx]
        iterations += 1
      return iterations

    total_iterations = 0
```

```
    for _ in range(1000):
      x_train, y_train = create_PLA_points()
      x_train_bias = np.hstack((np.ones((x_train.shape[0], 1)), x_train))
      x_transpose = x_train_bias.T
      inv = np.linalg.inv(np.dot(x_transpose, x_train_bias))
      w = np.dot(np.dot(inv, x_transpose), y_train)
      total_iterations += PLA(w, x_train, y_train)

    print(total_iterations / 1000)
```

```
3.568
```

8. The average in-sample error without transformation is ~0.5, according to my code. [d]

```
[6] def classify_nonlinear(point):
      x1 = point[0]
      x2 = point[1]
      val = x1**2 + x2**2 - 0.6
      if val == 0:
        return 0
      elif val < 0:
        return -1
      return 1

    def create_nonlinear_points():
      X = []
      Y = []
      X = np.random.uniform(-1.0, 1.0, (1000, 2))
      Y = np.array([classify_nonlinear(point) for point in X])
      random_indices = np.random.choice(np.arange(1000), size=100, replace=False)
      for i in random_indices:
        Y[i] *= -1
      return X, Y

    nonlinear_errors = []
    for _ in range(1000):
      X, Y = create_nonlinear_points()
      X_with_bias = np.hstack((np.ones((X.shape[0], 1)), X))
      x_transpose = X_with_bias.T
      inv = np.linalg.inv(np.dot(x_transpose, X_with_bias))
      w = np.dot(np.dot(inv, x_transpose), Y)
      nonlinear_errors.append(find_error(X_with_bias, Y, w))

    print(sum(nonlinear_errors) / len(nonlinear_errors))
```

```
0.5036800000000006
```

9. the first hypothesis agrees the most with mine, according to my code: => [a]

```
def create_transform_points():
  X = []
  Y = []
  X = np.random.uniform(-1.0, 1.0, (1000, 2))
  Y = np.array([classify_nonlinear(point) for point in X])
  noise_indices = np.random.choice(len(Y), 100, replace=False)
  Y[noise_indices] = -Y[noise_indices]
  x1 = X[:, 0]
  x2 = X[:, 1]
  bias = np.ones(X.shape[0])
  X_transformed = np.column_stack((bias, x1, x2, x1 * x2, x1 ** 2, x2 ** 2))
  return X, X_transformed, Y

def check_agreement(point, h):
  x1 = point[0]
  x2 = point[1]
  return np.sign(h[0] + x1 *h[1] + x2 *h[2] + x1*x2*h[3] + h[4] * x1**2 + h[5] * x2**2)


#now we want to see over 1000 runs, average probability
h1_agree = 0
h2_agree = 0
h3_agree = 0
h4_agree = 0
h5_agree = 0
```

```
[7] h1 = np.array([-1, -0.05, 0.08, 0.13, 1.5, 1.5])
    h2 = np.array([-1, -0.05, 0.08, 0.13, 1.5, 15])
    h3 = np.array([-1, -0.05, 0.08, 0.13, 15, 1.5])
    h4 = np.array([-1, -1.5, 0.08, 0.13, 0.05, 0.05])
    h5 = np.array([-1, -0.05, 0.08, 1.5, 0.15, 0.15])

    print(w)
    X_before, X, Y = create_transform_points()
    x_transpose = X.T
    inv = np.linalg.inv(np.dot(x_transpose, X))
    w = np.dot(np.dot(inv, x_transpose), Y)

    for _ in range(1000):
      x1 = np.random.uniform(-1.0, 1.0)
      x2 = np.random.uniform(-1.0, 1.0)
      w_sign =  check_agreement([x1, x2], w)
      if check_agreement([x1, x2], h1) == w_sign:
        h1_agree += 1
      if check_agreement([x1, x2], h2) == w_sign:
        h2_agree += 1
      if check_agreement([x1, x2], h3) == w_sign:
        h3_agree += 1
      if check_agreement([x1, x2], h4) == w_sign:
        h4_agree += 1
      if check_agreement([x1, x2], h5) == w_sign:
        h5_agree += 1

    print("hypothesis 1: ", h1_agree/1000)
    print("hypothesis 2: ", h2_agree/1000)
    print("hypothesis 3: ", h3_agree/1000)
    print("hypothesis 4: ", h4_agree/1000)
    print("hypothesis 5: ", h5_agree/1000)
```

```
[0.06570125 0.02862171 0.03411385]
hypothesis 1:  0.948
hypothesis 2:  0.674
hypothesis 3:  0.68
hypothesis 4:  0.625
hypothesis 5:  0.545
```

10. The out-of-sample error is closest to 0.1, according to my code: ⇒ [b]

```python
def generate_with_noise():
  X = []
  Y = []
  X = np.random.uniform(-1.0, 1.0, (1000, 2))
  Y = np.array([classify_nonlinear(point) for point in X])
  noise_indices = np.random.choice(len(Y), 100, replace=False)
  Y[noise_indices] = -Y[noise_indices]
  x1 = X[:, 0]
  x2 = X[:, 1]
  bias = np.ones(X.shape[0])
  X_transformed = np.column_stack((bias, x1, x2, x1 * x2, x1 ** 2, x2 ** 2))
  return X_transformed, Y

error_total = 0

for _ in range(1000):
  x_trans, y_noise = generate_with_noise()
  error_total += find_error(x_trans, y_noise, h1)

print(error_total / 1000)
```

```
0.14300700000000027
```