# Problem 1: SPAM, SPAM, HAM

In [27]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cvxopt
```

In [28]:
```
spam_train = np.loadtxt('spam_train.data', delimiter=',')
spam_validation = np.loadtxt('spam_validation.data', delimiter=',')
spam_test = np.loadtxt('spam_test.data', delimiter=',')
```

In [29]:
```
def preprocess(data):
    m, n = data.shape

    # X,y split
    X = data[:, :n-1]
    y = data[:, n-1:]

    # Set y = -1
    y = np.apply_along_axis(lambda x: -1 if x == 0 else 1, 1, y).reshape(-1, 1)

    return X,y
```

In [30]:
```
X_train, y_train = preprocess(spam_train)
X_validate, y_validate = preprocess(spam_validation)
X_test, y_test = preprocess(spam_test)
```

In [31]:
```
def get_accuracy(X, y, w, b, λ_ =None, y_ = None, X_= None, σ2=None):
    def gaussian_kernel(x, y, σ2):
        return np.exp(-np.linalg.norm(x-y)**2 / (2 * σ2))
    m, n = X.shape
    if w is not None:
        z = np.dot(X,w) + b
        f = (y * z) > 0
    else:
        print(f"Computing accuracy for σ2 = {σ2}")
        y_predict = np.zeros(m)
        for i in range(m):
            wx = 0
            for λ, sl, sv in zip(λ_, y_, X_):
                wx += λ * sl * gaussian_kernel(X[i], sv, σ2)
            y_predict[i] = wx
        y_predict + b
        f = (y_predict * y.ravel()) > 0
    return np.sum(f.astype('float32')) * 100/m
```

In [32]:
```
# Primal problem
def SVM_primal(X, Y, c):
    print(f"Computing c = {c}")
    m, n = X.shape

    P = np.zeros((m+n+1, m+n+1))
    P[:n,:n] = np.eye(n,n)
    P = cvxopt.matrix(P)

    q = np.zeros((m+n+1, 1))
    q[n:m+n,0] = c
    q = cvxopt.matrix(q)

    G = np.zeros((2*m, m+n+1))
    for i in range(m):
        for j in range(n):
            G[i][j] = -1 * Y[i] * X[i][j]
        G[i][n+i] = -1
        G[i][m+n] = -1 * Y[i]
        G[m+i][n+i] = -1
    G = cvxopt.matrix(G)

    h = np.zeros((2*m, 1))
    h[:m,0] = -1
    h = cvxopt.matrix(h)

    # CVXOPT Solver
    solution = cvxopt.solvers.qp(P, q, G, h)
    sol = np.array(solution['x'])
    w = sol[:n]
    b = sol[m+n]
    return w, b


# Dual problem
def SVM_dual(X, Y, c, σ2):

    print(f"Computing c = {c} and variance = {σ2}")
    m, n = X.shape

    # Create Gram matrix
    K = np.zeros((m, m))
    # Use gaussian kernel
    X_sq = -2 * np.dot(X, X.T)
    X_sq += (X ** 2).sum(axis=1).reshape(-1, 1)
    X_sq += (X ** 2).sum(axis=1)

    K = X_sq / (-2 * σ2)
    np.exp(K, K)
    # P = Combination of Yi Yj Xi Xj
    P = cvxopt.matrix(np.outer(Y,Y) * K)
    q = cvxopt.matrix(np.ones(m) * -1)

    # Constraints λY = 0
    A = cvxopt.matrix(Y, (1,m), 'd')
    b = cvxopt.matrix(0.0)
```

```
    # λ >= 0
    lhs = np.diag(np.ones(m) * -1)
    lhs2 = np.identity(m)
    G = cvxopt.matrix(np.vstack((lhs, lhs2)))
    rhs = np.zeros(m)
    rhs2 = np.ones(m) * c
    h = cvxopt.matrix(np.hstack((rhs, rhs2)))

    # CVXOPT Solver
    solution = cvxopt.solvers.qp(P, q, G, h, A, b)

    # Solver produces λ
    λ = np.ravel(solution['x'])
    n_λ = len(λ)
    sv = λ > 1e-5
    idx = np.arange(n_λ)[sv]
    l = λ[sv]
    support_labels = Y[sv]
    support_vectors = X[sv]

    b = 0.0
    for n in range(len(l)):
        b += support_labels[n]
        b -= np.sum(l * support_labels * K[idx[n],sv])
    b /= len(l)

    return b, l, support_labels, support_vectors
```

In [33]:
```
def run_svm(c_list, σ2_list = [None], primal = True):
    data = {
        'c': [],
        'variance': [],
        'Training Data Accuracy': [],
        'Validation Data Accuracy': []
    }
    best_c, best_σ2, best_validation_acc, best_train_acc = c_list[0], σ2_list[0],
    best_w, best_b = [], 0
    for c in c_list:
        for σ2 in σ2_list:
            data['c'].append(c)
            data['variance'].append(σ2)
            w, b, λ, support_labels, sv = None, None, None, None, None
            if primal:
                w, b = SVM_primal(X_train, y_train, c)
                train_acc = get_accuracy(X_train, y_train, w, b)
                validation_acc = get_accuracy(X_validate, y_validate, w, b)
            else:
                w = None
                b, λ, support_labels, sv = SVM_dual(X_train, y_train ,c,σ2)
                train_acc = get_accuracy(X_train, y_train, None, b, λ_ = λ, y_ =
                validation_acc = get_accuracy(X_validate, y_validate, None, b, λ_
            data['Training Data Accuracy'].append(train_acc)
            data['Validation Data Accuracy'].append(validation_acc)
            if validation_acc > best_validation_acc or (validation_acc == best_va
                best_validation_acc = validation_acc
                best_train_acc = train_acc
                best_c = c
                best_σ2 = σ2
                best_w = w
                best_b = b
                best_λ = λ
                best_sv = sv
                best_sv_y = support_labels
    df = pd.DataFrame.from_dict(data)
    if primal:
        test_acc = get_accuracy(X_test, y_test, best_w, best_b)
    else:
        test_acc = get_accuracy(X_train, y_train, None, best_b, λ_ = best_λ, y_ =
    df['Testing Data Accuracy'] = df.apply(lambda row: test_acc if row['c'] == be
    if best_σ2 is None:
        df.drop(columns=['variance'], inplace=True)
    return df
```

**1. Primal SVMs**

- Using gradient descent or quadratic programming, apply the SVM with slack formulation to train a classifier for each choice of $c \in 1, 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8$ without using any feature maps.
- What is the accuracy of the learned classifier on the training set for each value of c?
- Use the validation set to select the best value of c. What is the accuracy on the validation set for each value of c?

- Report the accuracy on the test set for the selected classifier.

In [34]:
```
c_list = [1,10,10**2,10**3,10**4,10**5,10**6,10**7,10**8]
df_primal = run_svm(c_list, primal=True)
```

```
Computing c = 1
     pcost       dcost       gap    pres   dres
 0: -1.8124e+03  1.0398e+04  8e+04  6e+00  4e+04
 1:  5.6137e+03 -7.3332e+03  2e+04  1e+00  7e+03
 2:  3.7452e+03 -2.4540e+03  7e+03  4e-01  2e+03
 3:  2.3041e+03 -9.9333e+02  4e+03  2e-01  1e+03
 4:  1.6237e+03 -3.8918e+02  2e+03  1e-01  6e+02
 5:  1.3035e+03 -9.5546e+01  2e+03  6e-02  4e+02
 6:  1.1640e+03  1.5815e+01  1e+03  4e-02  3e+02
 7:  1.1113e+03  8.9810e+01  1e+03  3e-02  2e+02
 8:  1.0293e+03  1.6990e+02  9e+02  2e-02  1e+02
 9:  8.2001e+02  2.9315e+02  5e+02  1e-02  6e+01
10:  7.1530e+02  3.5051e+02  4e+02  6e-03  4e+01
11:  6.2597e+02  3.9397e+02  2e+02  3e-03  2e+01
12:  5.6753e+02  4.2284e+02  1e+02  2e-03  1e+01
13:  5.3370e+02  4.3939e+02  1e+02  1e-03  6e+00
14:  5.1168e+02  4.5038e+02  6e+01  5e-04  3e+00
15:  4.8730e+02  4.6134e+02  3e+01  3e-05  2e-01
16:  4.7777e+02  4.6801e+02  1e+01  7e-06  4e-02
```

In [35]:
```
df_primal.style.apply(lambda x: ['background: lightgreen' if not np.isnan(x['Test
```

Out[35]:

| | c | Training Data Accuracy | Validation Data Accuracy | Testing Data Accuracy |
|---|---|---|---|---|
| **0** | 1 | 94.466667 | 93.500000 | nan |
| **1** | 10 | 94.733333 | 93.875000 | nan |
| **2** | 100 | 94.866667 | 93.875000 | 62.172285 |
| **3** | 1000 | 94.833333 | 93.750000 | nan |
| **4** | 10000 | 94.833333 | 93.750000 | nan |
| **5** | 100000 | 94.833333 | 93.750000 | nan |
| **6** | 1000000 | 94.833333 | 93.750000 | nan |
| **7** | 10000000 | 94.833333 | 93.750000 | nan |
| **8** | 100000000 | 94.833333 | 93.750000 | nan |

### 2. Dual SVMs with Gaussian Kernels

- Using quadratic programming, apply the dual of the SVM with slack formulation to train a classifier for each choice of c $c \in 1, 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8$ using a Gaussian kernel with $\sigma^2 \in .1, 1, 10, 100, 1000.$
- What is the accuracy of the learned classifier on the training set for each pair of c and σ?
- Use the validation set to select the best value of c and σ. What is the accuracy on the validation set for each pair of c and σ?
- Report the accuracy on the test set for the selected classifier.

In [10]:
```
c_list = [1,10,10**2,10**3,10**4,10**5,10**6,10**7,10**8]
σ2_list = [.1,1,10,100,1000]
df_dual = run_svm(c_list, σ2_list, primal=False)
```

```
Computing c = 1 and variance = 0.1
     pcost       dcost       gap    pres   dres
 0: -1.1526e+03 -5.7019e+03  1e+04  2e+00  4e-16
 1: -1.1153e+03 -3.2841e+03  2e+03  9e-13  2e-16
 2: -1.2019e+03 -1.4313e+03  2e+02  2e-12  1e-16
 3: -1.2565e+03 -1.2861e+03  3e+01  2e-12  1e-16
 4: -1.2645e+03 -1.2702e+03  6e+00  6e-12  6e-17
 5: -1.2662e+03 -1.2669e+03  8e-01  2e-12  6e-17
 6: -1.2663e+03 -1.2665e+03  2e-01  5e-12  5e-17
 7: -1.2663e+03 -1.2664e+03  4e-02  5e-13  5e-17
 8: -1.2663e+03 -1.2663e+03  1e-02  8e-12  6e-17
 9: -1.2663e+03 -1.2663e+03  5e-03  5e-12  7e-17
10: -1.2663e+03 -1.2663e+03  8e-04  1e-12  5e-17
Optimal solution found.
Computing c = 1 and variance = 1
     pcost       dcost       gap    pres   dres
 0: -1.1106e+03 -5.6547e+03  1e+04  2e+00  6e-16
 1: -1.0753e+03 -3.2369e+03  2e+03  2e-13  3e-16
 2: -1.1581e+03 -1.3869e+03  2e+02  8e-13  2e-16
```

In [11]:
```
df_dual.style.apply(lambda x: ['background: lightgreen' if not np.isnan(x['Testir
```

Out[11]:

| | c | variance | Training Data Accuracy | Validation Data Accuracy | Testing Data Accuracy |
|---|---|---|---|---|---|
| **0** | 1 | 0.100000 | 99.974000 | 19.623000 | nan |
| **1** | 1 | 1.000000 | 99.932000 | 26.389000 | nan |
| **2** | 1 | 10.000000 | 98.133333 | 68.755000 | nan |
| **3** | 1 | 100.000000 | 90.100000 | 79.625000 | nan |
| **4** | 1 | 1000.000000 | 83.233333 | 79.375000 | nan |
| **5** | 10 | 0.100000 | 99.973333 | 19.755000 | nan |
| **6** | 10 | 1.000000 | 99.973333 | 26.125000 | nan |
| **7** | 10 | 10.000000 | 99.773333 | 69.125000 | nan |
| **8** | 10 | 100.000000 | 97.736667 | 83.125000 | nan |
| **9** | 10 | 1000.000000 | 93.800000 | 85.875000 | nan |
| **10** | 100 | 0.100000 | 99.973333 | 19.125000 | nan |
| **11** | 100 | 1.000000 | 99.433333 | 26.125000 | nan |
| **12** | 100 | 10.000000 | 99.733333 | 69.125000 | nan |
| **13** | 100 | 100.000000 | 99.066667 | 84.125000 | nan |
| **14** | 100 | 1000.000000 | 97.033333 | 90.875000 | nan |
| **15** | 1000 | 0.100000 | 99.433333 | 19.125000 | nan |
| **16** | 1000 | 1.000000 | 99.433333 | 26.125000 | nan |
| **17** | 1000 | 10.000000 | 99.900000 | 68.125000 | nan |
| **18** | 1000 | 100.000000 | 99.966667 | 84.125000 | nan |
| **19** | 1000 | 1000.000000 | 98.900000 | 91.375000 | nan |
| **20** | 10000 | 0.100000 | 99.433333 | 19.125000 | nan |
| **21** | 10000 | 1.000000 | 99.433333 | 26.125000 | nan |
| **22** | 10000 | 10.000000 | 99.900000 | 68.125000 | nan |
| **23** | 10000 | 100.000000 | 99.533333 | 84.125000 | nan |
| **24** | 10000 | 1000.000000 | 99.000000 | 92.125000 | 81.125000 |
| **25** | 100000 | 0.100000 | 99.433333 | 19.125000 | nan |
| **26** | 100000 | 1.000000 | 99.433333 | 26.125000 | nan |
| **27** | 100000 | 10.000000 | 99.900000 | 68.125000 | nan |
| **28** | 100000 | 100.000000 | 99.866667 | 84.250000 | nan |
| **29** | 100000 | 1000.000000 | 99.433333 | 91.125000 | nan |
| **30** | 1000000 | 0.100000 | 99.433333 | 19.125000 | nan |
| **31** | 1000000 | 1.000000 | 99.433333 | 26.125000 | nan |
| **32** | 1000000 | 10.000000 | 99.900000 | 67.125000 | nan |
| **33** | 1000000 | 100.000000 | 99.533333 | 73.000000 | nan |

| | c | variance | Training Data Accuracy | Validation Data Accuracy | Testing Data Accuracy |
|---|---|---|---|---|---|
| **34** | 1000000 | 1000.000000 | 89.366667 | 72.125000 | nan |
| **35** | 10000000 | 0.100000 | 99.433333 | 19.125000 | nan |
| **36** | 10000000 | 1.000000 | 99.433333 | 26.125000 | nan |
| **37** | 10000000 | 10.000000 | 99.900000 | 68.125000 | nan |
| **38** | 10000000 | 100.000000 | 84.533333 | 69.000000 | nan |
| **39** | 10000000 | 1000.000000 | 87.200000 | 70.125000 | nan |
| **40** | 100000000 | 0.100000 | 99.433333 | 19.125000 | nan |
| **41** | 100000000 | 1.000000 | 99.433333 | 26.125000 | nan |
| **42** | 100000000 | 10.000000 | 99.900000 | 68.125000 | nan |
| **43** | 100000000 | 100.000000 | 99.533333 | 71.000000 | nan |
| **44** | 100000000 | 1000.000000 | 81.266667 | 53.125000 | nan |

### 3. k-Nearest Neighbors

- What is the accuracy of the k-nearest neighbor classifier for k = 1,5,11,15,21?

In [33]:

```python
from collections import defaultdict

class kNN:

    def __init__(self, neighbors=1):
        self.train_data = []
        self.labels = []
        self.m = 0
        self.n = 0
        self.k = neighbors

    def fit(self, X_train, y_train):
        self.train_mean = X_train.mean(axis=0)
        self.train_std = X_train.std(axis=0)
        self.train_data = X_train
        self.train_data_normed = (self.train_data - self.train_mean) / self.trair
        self.m, self.n = self.train_data.shape
        self.labels = y_train

    def get_distances(self, X_test):
        distances = -2 * self.train_data_normed.dot(X_test.T) + np.sum(X_test**2,
        distances[distances < 0] = 0
        return distances

    def predict(self, X_test):
        X_test_normed = (X_test - self.train_mean) / self.train_std
        distances = self.get_distances(X_test_normed)
        idx = np.argsort(distances, axis=0)
        idx = idx[0:self.k, :]
        m, n = idx.shape
        labels = self.labels.ravel()
        y_pred = np.zeros((X_test.shape[0], 1))
        for col in range(n):
            classes = defaultdict(int)
            for row in range(m):
                label = labels[idx[row, col]]
                classes[label] += 1
            # Get the majority class
            y_pred[col] = max(classes, key=classes.get)
        return y_pred

    @staticmethod
    def get_accuracy(y_pred, y_test):
        return np.mean(y_pred.flatten() == y_test.flatten()) * 100
```

In [34]:
```python
def run_kNN(k_list):
    data = {
        'k': [],
        'Validation Accuracy': [],
        'Test data Accuracy': []
    }
    for k in k_list:
        data['k'].append(k)
        classifier = kNN(k)
        classifier.fit(X_train, y_train)
        y_validate_pred = classifier.predict(X_validate)
        data['Validation Accuracy'].append(kNN.get_accuracy(y_validate_pred, y_va
        y_test_pred = classifier.predict(X_test)
        data['Test data Accuracy'].append(kNN.get_accuracy(y_test_pred, y_test))

    return pd.DataFrame.from_dict(data)
```

In [35]:
```python
k_list = [1,5,11,15,21]
df = run_kNN(k_list)
display(df)
```

| | k | Validation Accuracy | Test data Accuracy |
|---|---|---|---|
| **0** | 1 | 88.500 | 72.659176 |
| **1** | 5 | 89.625 | 70.786517 |
| **2** | 11 | 88.500 | 72.159800 |
| **3** | 15 | 86.375 | 71.535581 |
| **4** | 21 | 86.500 | 70.911361 |

In [10]:
```python
m, n = X_train.shape
X = X_train
c = 10
Y = y_train
P = np.zeros((m+n+1, m+n+1))
P[:n,:n] = np.eye(n,n)
q = np.zeros((m+n+1, 1))
q[n:m+n,0] = c
G = np.zeros((2*m, m+n+1))
for i in range(m):
    for j in range(n):
        G[i][j] = -1 * Y[i] * X[i][j]
    G[i][n+i] = -1
    G[i][m+n] = -1 * Y[i]
    G[m+i][n+i] = -1

h = np.zeros((2*m, 1))
h[:m,0] = -1
```

In [26]:
```
h2 = np.zeros((m*2, 1))
h2[:m,0] = -1
np.mean(h2.ravel() == h.ravel())
```

Out[26]:
```
1.0
```

**4. Which of these approaches (if any) should be preferred for this classification task? Explain**

SVM with gaussian kernel should be prefered for this classification task as it has higher test data accuracy of 90.38% compared to kNN model which has average test data accuracy ~70%. SVM performs better in Higher dimensions compared to kNN.