Machine Learning HWS24

Assignment 2

Cagan Yigit Deliktas, cdelikta, 1979012

Nursultan Mamatov, nmamatov, 1983726

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random
import numpy.linalg
import scipy.io
import scipy.stats
import sklearn.metrics
%matplotlib inline

# setup plotting
from IPython import get_ipython
import psutil
inTerminal = not "IPKernelApp" in get_ipython().config
inJupyterNb = any(filter(lambda x: x.endswith("jupyter-notebook"), psutil.Process().parent().cmdline()))
get_ipython().run_line_magic("matplotlib", "" if inTerminal else "notebook" if inJupyterNb else "widget")
def nextplot():
    if inTerminal:
        plt.clf()     # this clears the current plot
    else:
        plt.figure()  # this creates a new plot
```

Load the data

In [2]:
```
data = scipy.io.loadmat("data/spamData.mat")
X = data["Xtrain"]
N = X.shape[0]
D = X.shape[1]
Xtest = data["Xtest"]
Ntest = Xtest.shape[0]
y = data["ytrain"].squeeze().astype(int)
ytest = data["ytest"].squeeze().astype(int)

features = np.array(
    [
        "word_freq_make",
        "word_freq_address",
        "word_freq_all",
        "word_freq_3d",
        "word_freq_our",
        "word_freq_over",
        "word_freq_remove",
        "word_freq_internet",
        "word_freq_order",
        "word_freq_mail",
        "word_freq_receive",
        "word_freq_will",
        "word_freq_people",
        "word_freq_report",
        "word_freq_addresses",
        "word_freq_free",
        "word_freq_business",
        "word_freq_email",
        "word_freq_you",
        "word_freq_credit",
        "word_freq_your",
        "word_freq_font",
        "word_freq_000",
        "word_freq_money",
        "word_freq_hp",
        "word_freq_hpl",
        "word_freq_george",
        "word_freq_650",
        "word_freq_lab",
        "word_freq_labs",
        "word_freq_telnet",
        "word_freq_857",
        "word_freq_data",
```

```
        "word_freq_415",
        "word_freq_85",
        "word_freq_technology",
        "word_freq_1999",
        "word_freq_parts",
        "word_freq_pm",
        "word_freq_direct",
        "word_freq_cs",
        "word_freq_meeting",
        "word_freq_original",
        "word_freq_project",
        "word_freq_re",
        "word_freq_edu",
        "word_freq_table",
        "word_freq_conference",
        "char_freq_;",
        "char_freq_(",
        "char_freq_[",
        "char_freq_!",
        "char_freq_$",
        "char_freq_#",
        "capital_run_length_average",
        "capital_run_length_longest",
        "capital_run_length_total",
    ]
)
```

In [3]:
```
import pandas as pd
```

In [4]:
```
test_df = pd.concat([pd.DataFrame(Xtest, columns=features.tolist()), pd.DataFrame(ytest, columns=['Spam'])], ax:
display(test_df.head())
print(test_df.shape)
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_i
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	
1	0.06	0.00	0.71	0.0	1.23	0.19	0.19	
2	0.15	0.00	0.46	0.0	0.61	0.00	0.30	
3	0.06	0.12	0.77	0.0	0.19	0.32	0.38	
4	0.00	0.69	0.34	0.0	0.34	0.00	0.00	

5 rows × 58 columns

```
(1536, 58)
```

In [5]:
```
train_df = pd.concat([pd.DataFrame(X, columns=features.tolist()), pd.DataFrame(y, columns=['Spam'])], axis=1)
display(train_df.head())
print(train_df.shape)
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_i
0	0.21	0.28	0.5	0.0	0.14	0.28	0.21	
1	0.00	0.00	0.0	0.0	0.63	0.00	0.31	
2	0.00	0.00	0.0	0.0	0.63	0.00	0.31	
3	0.00	0.00	0.0	0.0	1.85	0.00	0.00	
4	0.00	0.00	0.0	0.0	1.92	0.00	0.00	

5 rows × 58 columns

```
(3065, 58)
```

In [6]:
```
train_df.describe()
```

Out[6]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_
count	3065.000000	3065.000000	3065.000000	3065.000000	3065.000000	3065.000000	3065.000000	
mean	0.110819	0.228486	0.274153	0.062969	0.317788	0.095755	0.113546	
std	0.327252	1.373834	0.484063	1.334772	0.663570	0.260613	0.373958	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
75%	0.000000	0.000000	0.410000	0.000000	0.390000	0.000000	0.000000	
max	4.540000	14.280000	5.100000	42.810000	9.090000	3.570000	7.270000	

8 rows × 58 columns

1. Dataset Statistics

In [7]:

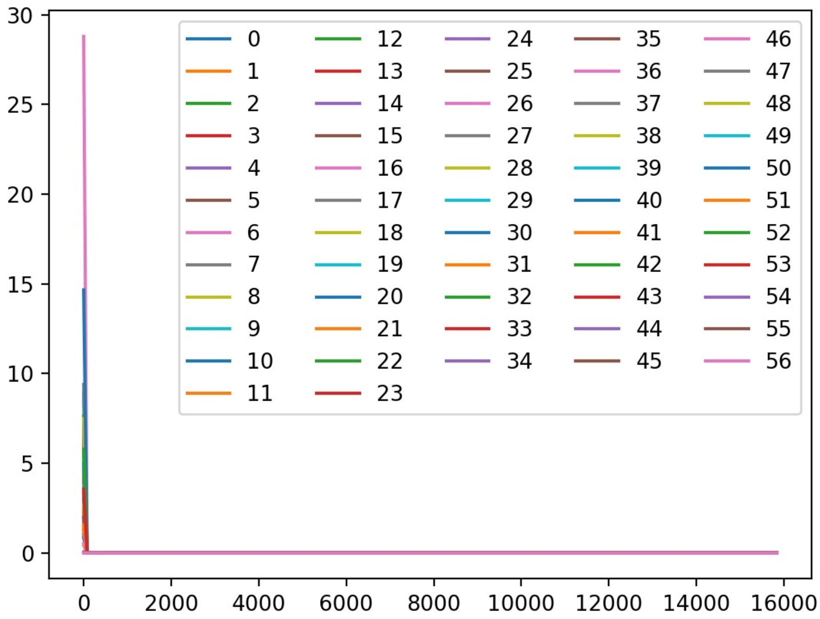
```
# look some dataset statistics
scipy.stats.describe(X)
```

```
Out[7]: DescribeResult(nobs=3065, minmax=(array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 1., 1., 1.]), array([4.5400e+00, 1.4280e+01, 5.1000e+00, 4.2810e+01, 9.0900e+00,
               3.5700e+00, 7.2700e+00, 1.1110e+01, 3.3300e+00, 1.8180e+01,
               2.0000e+00, 9.6700e+00, 5.5500e+00, 5.5500e+00, 2.8600e+00,
               1.0160e+01, 7.1400e+00, 9.0900e+00, 1.8750e+01, 6.3200e+00,
               1.1110e+01, 1.7100e+01, 5.4500e+00, 9.0900e+00, 2.0000e+01,
               1.4280e+01, 3.3330e+01, 4.7600e+00, 1.4280e+01, 4.7600e+00,
               4.7600e+00, 4.7600e+00, 1.8180e+01, 4.7600e+00, 2.0000e+01,
               7.6900e+00, 6.8900e+00, 7.4000e+00, 9.7500e+00, 4.7600e+00,
               7.1400e+00, 1.4280e+01, 3.5700e+00, 2.0000e+01, 2.1420e+01,
               1.6700e+01, 2.1200e+00, 1.0000e+01, 4.3850e+00, 9.7520e+00,
               4.0810e+00, 3.2478e+01, 6.0030e+00, 1.9829e+01, 1.1025e+03,
               9.9890e+03, 1.5841e+04])), mean=array([1.10818923e-01, 2.28486134e-01, 2.74153344e-01, 6.29690049e-02,
               3.17787928e-01, 9.57553018e-02, 1.13546493e-01, 1.07216966e-01,
               8.89233279e-02, 2.41719413e-01, 5.81305057e-02, 5.37432300e-01,
               9.26231648e-02, 4.96639478e-02, 5.07210440e-02, 2.35334421e-01,
               1.47197390e-01, 1.86600326e-01, 1.66121044e+00, 7.63066884e-02,
               8.19592170e-01, 1.22727569e-01, 1.02006525e-01, 8.90799347e-02,
               5.29800979e-01, 2.62071778e-01, 7.71507341e-01, 1.14323002e-01,
               1.09487765e-01, 9.92952692e-02, 6.28156607e-02, 4.90342577e-02,
               9.27471452e-02, 4.96019576e-02, 1.02156607e-01, 9.93050571e-02,
               1.43285481e-01, 1.24274062e-02, 7.55921697e-02, 6.60456770e-02,
               4.63360522e-02, 1.32176183e-01, 4.88580750e-02, 7.11876020e-02,
               3.06590538e-01, 1.79794454e-01, 5.28874388e-03, 3.13768352e-02,
               3.79543230e-02, 1.38396411e-01, 1.81830343e-02, 2.65470799e-01,
               7.91275693e-02, 5.34218597e-02, 4.90062936e+00, 5.26750408e+01,
               2.82203915e+02]), variance=array([1.07094140e-01, 1.88742036e+00, 2.34317437e-01, 1.78161723e+00,
               4.40325719e-01, 6.79193461e-02, 1.39844435e-01, 1.72001423e-01,
               6.97247542e-02, 4.69800274e-01, 3.58302179e-02, 7.59167719e-01,
               9.28365241e-02, 8.26118648e-02, 7.00470321e-02, 4.29393369e-01,
               2.00636301e-01, 2.92991898e-01, 3.18992370e+00, 1.65626303e-01,
               1.44315254e+00, 1.01505046e+00, 1.19749530e-01, 1.43862796e-01,
               2.45800502e+00, 7.38036013e-01, 1.13920029e+01, 2.31010973e-01,
               4.31507668e-01, 1.90528093e-01, 1.24671084e-01, 1.07425177e-01,
               2.95159161e-01, 1.07745599e-01, 3.08154062e-01, 1.67896547e-01,
               1.85791650e-01, 4.34829439e-02, 1.42525114e-01, 1.16865102e-01,
               1.50361473e-01, 6.09903912e-01, 5.73945833e-02, 3.19259425e-01,
               1.01935877e+00, 8.17471270e-01, 4.63438951e-03, 7.50333517e-02,
               5.54612799e-02, 7.77968333e-02, 1.48045497e-02, 7.59181612e-01,
               6.74541224e-02, 2.69600271e-01, 7.42311765e+02, 4.86573219e+04,
               3.68952901e+05]), skewness=array([ 5.92257918,  9.5555492 ,  2.94110789, 27.15035267,  4.22000271,
                4.55490419,  6.21454549, 10.63604439,  4.44795353,  9.63368819,
                5.1601559 ,  3.12797362,  7.99555783, 10.07103212,  6.44051978,
                5.9017492 ,  5.71193665,  5.63845456,  1.6918398 ,  8.05102821,
                2.36131511,  9.70708774,  5.74851972, 13.62929854,  5.51200726,
                5.77490458,  5.72163481,  5.84582426, 11.30526457,  6.67894971,
                8.78006633, 10.35563132, 16.1291286 , 10.31146394, 17.98980105,
                7.86085564,  5.29526945, 27.69555992, 10.51869112,  9.12514394,
               12.60532735,  9.42688905,  7.88762618, 19.69945392,  9.63372543,
                8.97501221, 18.94255005, 20.98217881, 14.12336521, 16.36382061,
               21.32440567, 21.32959254, 10.88427173, 26.25786993, 27.34951229,
               31.14016596,  9.80477376]), kurtosis=array([  51.71558405,   93.89016173,   13.18839908,  785.40163828,
                 28.69487647,   31.20576951,   66.53150801,  198.68010939,
                 28.29530115,  185.40607771,   34.48800593,   15.18712484,
                109.66544541,  138.05561341,   44.19188958,   55.62892   ,
                 47.49151277,   52.75647121,    6.32523058,   77.87379384,
                  8.48736408,  103.7022867 ,   49.37553046,  272.09125904,
                 42.43992409,   49.41302953,   33.63974328,   39.86629858,
                166.19735746,   53.12216402,   91.72439904,  124.79234055,
                433.42661801,  123.97955409,  555.16708959,   86.72460731,
                 43.92486688,  865.39968623,  181.33012173,  100.87592785,
                189.11563172,  111.21705016,   81.96093958,  567.75150773,
                147.5283386 ,  107.79164424,  445.8361165 ,  634.57001982,
                228.75884956,  499.07842266,  588.19774644,  688.05527222,
                184.31757803,  851.48819158,  954.59095344, 1348.49464105,
                183.78053905]))
```

```
In [8]: scipy.stats.describe(y)
```

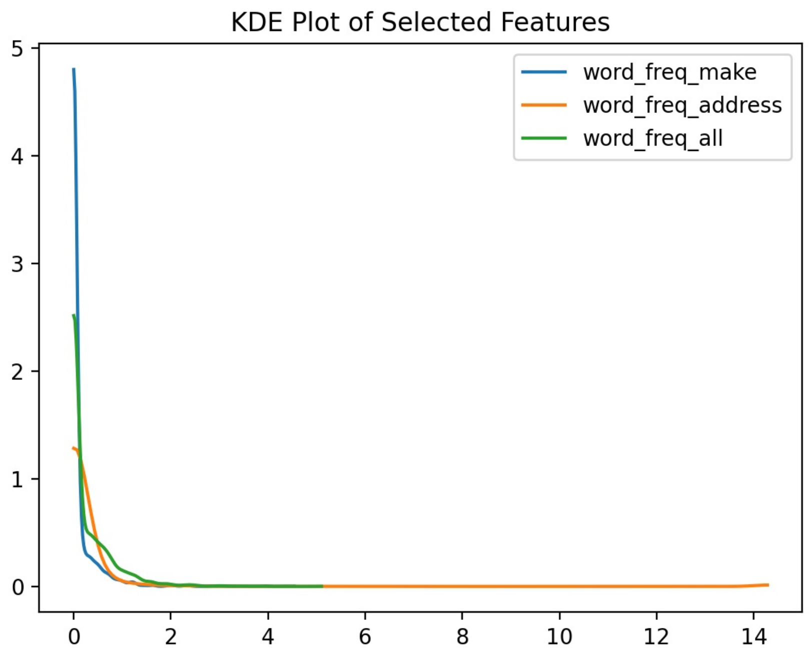
```
Out[8]: DescribeResult(nobs=3065, minmax=(0, 1), mean=0.39738988580750406, variance=0.23954932085067235, skewness=0.419
        36632478193103, kurtosis=-1.824131885638896)
```

```
In [9]: # plot the distribution of all features
        nextplot()
        densities = [scipy.stats.gaussian_kde(X[:, j]) for j in range(D)]
        xs = np.linspace(0, np.max(X), 200)
        for j in range(D):
            plt.plot(xs, densities[j](xs), label=j)
        plt.legend(ncol=5)
```



Out[9]: <matplotlib.legend.Legend at 0x15db23490>

In [10]:
```
# this plots is not really helpful; go now explore further
# YOUR CODE HERE

selected_features = [0, 1, 2]

def plot_selected_features(X, feature_names, selected_features):
    """
    Plots kernel density estimates for the specified features.
    """
    nextplot()

    for j in selected_features:
        density = scipy.stats.gaussian_kde(X[:, j])
        xs = np.linspace(0, np.max(X[:, j]), 200)
        plt.plot(xs, density(xs), label=feature_names[j])

    plt.legend(ncol=1)
    plt.title("KDE Plot of Selected Features")
    plt.show()

plot_selected_features(X, features, selected_features)
```



In [11]:
```
np.mean(X, axis=0)
```

Out[11]:
```
array([1.10818923e-01, 2.28486134e-01, 2.74153344e-01, 6.29690049e-02,
       3.17787928e-01, 9.57553018e-02, 1.13546493e-01, 1.07216966e-01,
       8.89233279e-02, 2.41719413e-01, 5.81305057e-02, 5.37432300e-01,
       9.26231648e-02, 4.96639478e-02, 5.07210440e-02, 2.35334421e-01,
       1.47197390e-01, 1.86600326e-01, 1.66121044e+00, 7.63066884e-02,
       8.19592170e-01, 1.22727569e-01, 1.02006525e-01, 8.90799347e-02,
       5.29800979e-01, 2.62071778e-01, 7.71507341e-01, 1.14323002e-01,
       1.09487765e-01, 9.92952692e-02, 6.28156607e-02, 4.90342577e-02,
       9.27471452e-02, 4.96019576e-02, 1.02156607e-01, 9.93050571e-02,
       1.43285481e-01, 1.24274062e-02, 7.55921697e-02, 6.60456770e-02,
       4.63360522e-02, 1.32176183e-01, 4.88580750e-02, 7.11876020e-02,
       3.06590538e-01, 1.79794454e-01, 5.28874388e-03, 3.13768352e-02,
       3.79543230e-02, 1.38396411e-01, 1.81830343e-02, 2.65470799e-01,
       7.91275693e-02, 5.34218597e-02, 4.90062936e+00, 5.26750408e+01,
       2.82203915e+02])
```

In [12]:
```
# Let's compute z-scores; create two new variables Xz and Xtestz.
mean_train = np.mean(X, axis=0)
std_train = np.std(X, axis=0)

Xz = (X - mean_train) / std_train
Xtestz = (Xtest - mean_train) / std_train
```

In [13]:
```
# Let's check. Xz and Xtestz refer to the normalized datasets just created. We
# will use them throughout.
print("mean train -- # should be all 0\n", np.mean(Xz, axis=0), '\n')  # should be all 0
print("var train -- # should be all 1\n", np.var(Xz, axis=0), '\n')  # should be all 1
print("mean test -- # what do you get here?\n",np.mean(Xtestz, axis=0), '\n')  # what do you get here?
print("var test -- \n", np.var(Xtestz, axis=0), '\n')

print("should be: 1925261.15\n", np.sum(Xz ** 3), '\n')  # should be: 1925261.15
```

```
mean train -- # should be all 0
 [ 1.85459768e-17  9.27298839e-18 -5.56379304e-17 -9.27298839e-18
  5.56379304e-17  3.70919536e-17  0.00000000e+00 -7.41839072e-17
  5.56379304e-17  0.00000000e+00 -1.85459768e-17 -2.43415945e-17
 -4.63649420e-17  1.85459768e-17  1.85459768e-17  3.70919536e-17
 -3.70919536e-17 -9.27298839e-17 -1.66913791e-16  9.27298839e-18
  1.85459768e-17  9.27298839e-18 -5.56379304e-17 -1.85459768e-17
 -6.49109188e-17 -3.70919536e-17 -1.85459768e-17  1.85459768e-17
 -2.78189652e-17  4.63649420e-17 -1.85459768e-17  5.56379304e-17
  0.00000000e+00 -1.85459768e-17  3.70919536e-17  1.85459768e-17
 -9.27298839e-18  4.63649420e-18  1.85459768e-17  9.27298839e-18
  2.31824710e-17 -2.78189652e-17 -9.27298839e-18  4.63649420e-18
 -9.27298839e-18 -9.27298839e-18  1.39094826e-17 -2.78189652e-17
 -3.70919536e-17 -6.49109188e-17  4.63649420e-18  3.70919536e-17
 -3.70919536e-17  9.27298839e-18 -9.27298839e-18  9.27298839e-18
 -7.41839072e-17]

var train -- # should be all 1
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1.]

mean test -- # what do you get here?
 [-5.73600192e-02 -3.37389835e-02  4.02481250e-02  5.51233798e-03
 -2.51229644e-02  1.67364997e-03  5.29785531e-03 -1.38875040e-02
  1.29802458e-02 -1.00804532e-02  2.68026912e-02  1.46804853e-02
  1.28455840e-02  9.34193448e-02 -1.71666713e-02  6.17841473e-02
 -3.08405298e-02 -1.02710095e-02  1.49139906e-03  6.82438979e-02
 -2.45179646e-02 -4.53675036e-03 -3.12737328e-03  4.09841941e-02
  3.76515934e-02  1.15494599e-02 -3.73018154e-03  6.55839018e-02
 -4.82178216e-02  2.44089391e-02  1.64408852e-02 -1.81514851e-02
  2.47142980e-02 -1.61248615e-02  1.75684573e-02 -1.33686432e-02
 -4.40153254e-02  1.11212504e-02  2.40959269e-02 -1.06211719e-02
 -2.06246544e-02  6.23149655e-04 -3.45073187e-02  4.24615929e-02
 -1.59254291e-02  9.77429328e-05  6.85319587e-03  5.38462415e-03
  7.89156240e-03  6.81007462e-03 -2.97234292e-02  1.23785037e-02
 -3.82610483e-02 -5.29891640e-02  3.19860888e-02 -6.82149671e-03
  5.35333143e-03]

var test --
 [0.61068019 0.64746339 1.25293677 1.2774661  1.08119249 1.31173762
 1.28697678 0.80611698 1.33973062 0.65533893 1.40034314 0.93450565
 0.92877323 2.0728468  0.86981179 2.75968123 0.94816223 0.88879741
 0.96502082 2.70171906 0.99741759 1.1098788  1.07414603 2.08336518
 1.40816544 1.19772845 0.9862879  1.76326753 0.44704368 1.28342341
 1.91457064 1.01476883 1.14073258 1.02208023 0.75850361 0.89687605
 0.89454052 1.35876298 1.97554069 1.14319113 0.60370645 0.89279613
 0.61835224 1.633395   1.01236044 1.04674566 1.76525404 1.2642542
 1.20646248 0.81912474 0.42556335 0.62984245 0.68863812 0.05099329
 2.06687781 0.34306778 0.98979083]

should be: 1925261.15
 1925261.1560010156
```

In [14]:

```python
# Explore the normalized data
# YOUR CODE HERE

train_df_z = pd.concat([pd.DataFrame(Xz, columns=features.tolist()), pd.DataFrame(y, columns=['Spam'])], axis=1
display(train_df_z.describe().round(3))

print('******************************')

test_df_z = pd.concat([pd.DataFrame(Xtestz, columns=features.tolist()), pd.DataFrame(ytest, columns=['Spam'])],
display(test_df_z.describe().round(3))
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_fr
count	3065.000	3065.000	3065.000	3065.000	3065.000	3065.000	3065.000	
mean	0.000	0.000	-0.000	-0.000	0.000	0.000	0.000	
std	1.000	1.000	1.000	1.000	1.000	1.000	1.000	
min	-0.339	-0.166	-0.566	-0.047	-0.479	-0.367	-0.304	
25%	-0.339	-0.166	-0.566	-0.047	-0.479	-0.367	-0.304	
50%	-0.339	-0.166	-0.566	-0.047	-0.479	-0.367	-0.304	
75%	-0.339	-0.166	0.281	-0.047	0.109	-0.367	-0.304	
max	13.537	10.230	9.971	32.031	13.222	13.333	19.140	

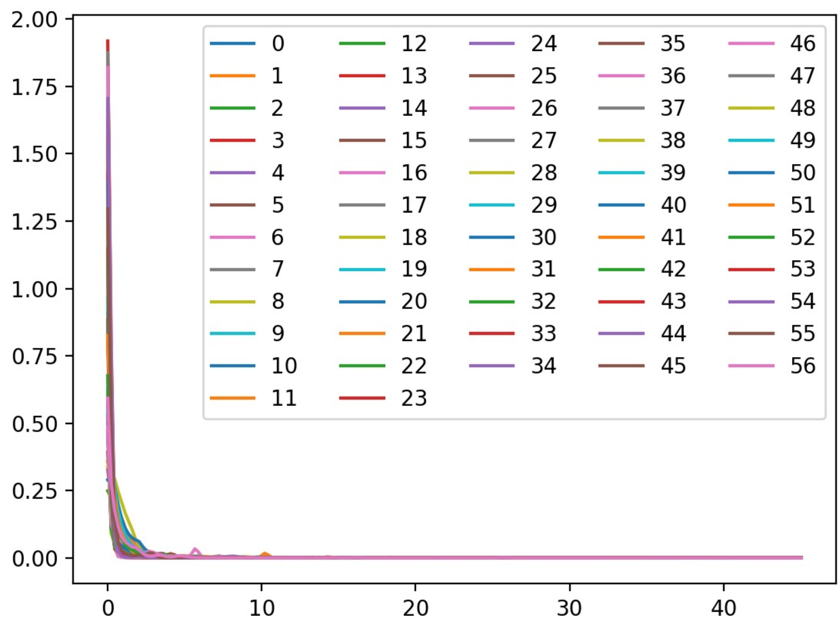
8 rows × 58 columns

****

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_fr
count	1536.000	1536.000	1536.000	1536.000	1536.000	1536.000	1536.000	
mean	-0.057	-0.034	0.040	0.006	-0.025	0.002	0.005	
std	0.782	0.805	1.120	1.131	1.040	1.146	1.135	
min	-0.339	-0.166	-0.566	-0.047	-0.479	-0.367	-0.304	
25%	-0.339	-0.166	-0.566	-0.047	-0.479	-0.367	-0.304	
50%	-0.339	-0.166	-0.566	-0.047	-0.479	-0.367	-0.304	
75%	-0.339	-0.166	0.322	-0.047	0.067	-0.367	-0.304	
max	6.599	10.230	8.814	31.971	14.593	22.198	19.140	

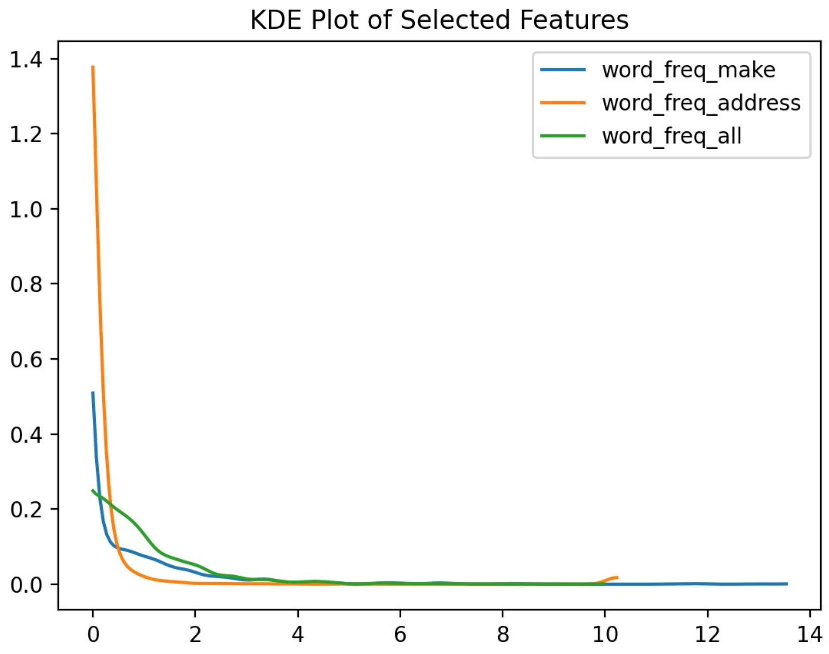
8 rows × 58 columns

In [15]:
```
## Redo the Kernel Density Plot
nextplot()
densities = [scipy.stats.gaussian_kde(Xz[:, j]) for j in range(D)]
xs = np.linspace(0, np.max(Xz), 200)
for j in range(D):
    plt.plot(xs, densities[j](xs), label=j)
plt.legend(ncol=5)
```



Out[15]: <matplotlib.legend.Legend at 0x15aefa090>

In [16]:
```
selected_features = [0, 1, 2]
plot_selected_features(Xz, features, selected_features)
```



2. Maximum Likelihood Estimation

Helper functions

In [17]:
```
def logsumexp(x):
    """Computes log(sum(exp(x)).

    Uses offset trick to reduce risk of numeric over- or underflow. When x is a
    1D ndarray, computes logsumexp of its entries. When x is a 2D ndarray,
    computes logsumexp of each column.

    Keyword arguments:
    x : a 1D or 2D ndarray
    """
    offset = np.max(x, axis=0)
    return offset + np.log(np.sum(np.exp(x - offset), axis=0))
```

In [18]:
```
# Define the logistic function. Make sure it operates on both scalars
# and vectors.
def sigma(x):
    # YOUR CODE HERE
    return 1 / (1 + np.exp(-x))
```

In [19]:
```
# this should give:
# [0.5, array([0.26894142, 0.5, 0.73105858])]
[sigma(0), sigma(np.array([-1, 0, 1]))]
```

Out[19]:
```
[0.5, array([0.26894142, 0.5       , 0.73105858])]
```

In [20]:
```
# Define the logarithm of the logistic function. Make sure it operates on both
# scalars and vectors. Perhaps helpful: isinstance(x, np.ndarray).
def logsigma(x):
    # YOUR CODE HERE
    # Check if x is a numpy array
    if isinstance(x, np.ndarray):
        return -np.log1p(np.exp(-x))
    else:
        return -np.log1p(np.exp(-x))
```

In [21]:
```
# this should give:
# [-0.69314718055994529, array([-1.31326169, -0.69314718, -0.31326169])]
[logsigma(0), logsigma(np.array([-1, 0, 1]))]
```

Out[21]:
```
[-0.6931471805599453, array([-1.31326169, -0.69314718, -0.31326169])]
```

2b Log-likelihood and gradient

In [22]:
```python
def l(y, X, w):
    """Log-likelihood of the logistic regression model.

    Parameters
    ----------
    y : ndarray of shape (N,)
        Binary labels (either 0 or 1).
    X : ndarray of shape (N,D)
        Design matrix.
    w : ndarray of shape (D,)
        Weight vector.
    """
    # YOUR CODE HERE
    z = X @ w
    log_likelihood = np.sum(y * logsigma(z) + (1 - y) * logsigma(-z))
    return log_likelihood
```

In [23]:
```python
# this should give:
# -47066.641667825766
l(y, Xz, np.linspace(-5, 5, D))
```

Out[23]:
```
-47066.641667825774
```

In [24]:
```python
def dl(y, X, w):
    """Gradient of the log-likelihood of the logistic regression model.

    Parameters
    ----------
    y : ndarray of shape (N,)
        Binary labels (either 0 or 1).
    X : ndarray of shape (N,D)
        Design matrix.
    w : ndarray of shape (D,)
        Weight vector.

    Returns
    -------
    ndarray of shape (D,)
    """
    # YOUR CODE HERE
    gradient = X.T @ (y - sigma(X @ w))
    return gradient
```

In [25]:
```python
# this should give:
# array([  551.33985842,   143.84116318,   841.83373606,   156.87237578,
#          802.61217579,   795.96202907,   920.69045803,   621.96516752,
#          659.18724769,   470.81259805,   771.32406968,   352.40325626,
#          455.66972482,   234.36600888,   562.45454038,   864.83981264,
#          787.19723703,   649.48042176,   902.6478154 ,   544.00539886,
#         1174.78638035,   120.3598967 ,   839.61141672,   633.30453444,
#         -706.66815087,  -630.2039816 ,  -569.3451386 ,  -527.50996698,
#         -359.53701083,  -476.64334832,  -411.60620464,  -375.11950586,
#         -345.37195689,  -376.22044258,  -407.31761977,  -456.23251936,
#         -596.86960184,  -107.97072355,  -394.82170044,  -229.18125598,
#         -288.46356547,  -362.13402385,  -450.87896465,  -277.03932676,
#         -414.99293368,  -452.28771693,  -167.54649092,  -270.9043748 ,
#         -252.20140951,  -357.72497343,  -259.12468742,   418.35938483,
#          604.54173228,    43.10390907,   152.24258478,   378.16731033,
#          416.12032881])
dl(y, Xz, np.linspace(-5, 5, D))
```

Out[25]:
```
array([ 551.33985842,  143.84116318,  841.83373606,  156.87237578,
        802.61217579,  795.96202907,  920.69045803,  621.96516752,
        659.18724769,  470.81259805,  771.32406968,  352.40325626,
        455.66972482,  234.36600888,  562.45454038,  864.83981264,
        787.19723703,  649.48042176,  902.6478154 ,  544.00539886,
       1174.78638035,  120.3598967 ,  839.61141672,  633.30453444,
       -706.66815087, -630.2039816 , -569.3451386 , -527.50996698,
       -359.53701083, -476.64334832, -411.60620464, -375.11950586,
       -345.37195689, -376.22044258, -407.31761977, -456.23251936,
       -596.86960184, -107.97072355, -394.82170044, -229.18125598,
       -288.46356547, -362.13402385, -450.87896465, -277.03932676,
       -414.99293368, -452.28771693, -167.54649092, -270.9043748 ,
       -252.20140951, -357.72497343, -259.12468742,  418.35938483,
        604.54173228,   43.10390907,  152.24258478,  378.16731033,
        416.12032881])
```

2c Gradient descent

In [26]:
```python
# you don't need to modify this function
def optimize(obj_up, theta0, nepochs=50, eps0=0.01, verbose=True):
    """Iteratively minimize a function.

    We use it here to run either gradient descent or stochastic gradient
    descent, using arbitrarly optimization criteria.

    Parameters
    ----------
    obj_up  : a tuple of form (f, update) containing two functions f and update.
              f(theta) computes the value of the objective function.
              update(theta,eps) performs an epoch of parameter update with step size
              eps and returns the result.
    theta0  : ndarray of shape (D,)
              Initial parameter vector.
    nepochs : int
              How many epochs (calls to update) to run.
    eps0    : float
              Initial step size.
    verbose : boolean
              Whether to print progress information.

    Returns
    -------
    A triple consisting of the fitted parameter vector, the values of the
    objective function after every epoch, and the step sizes that were used.
    """

    f, update = obj_up

    # initialize results
    theta = theta0
    values = np.zeros(nepochs + 1)
    eps = np.zeros(nepochs + 1)
    values[0] = f(theta0)
    eps[0] = eps0

    # now run the update function nepochs times
    for epoch in range(nepochs):
        if verbose:
            print(
                "Epoch {:3d}: f={:10.3f}, eps={:10.9f}".format(
                    epoch, values[epoch], eps[epoch]
                )
            )
        theta = update(theta, eps[epoch])

        # we use the bold driver heuristic
        values[epoch + 1] = f(theta)
        if values[epoch] < values[epoch + 1]:
            eps[epoch + 1] = eps[epoch] / 2.0
        else:
            eps[epoch + 1] = eps[epoch] * 1.05

    # all done
    if verbose:
        print("Result after {} epochs: f={}".format(nepochs, values[-1]))
    return theta, values, eps
```

In [27]:
```python
# define the objective and update function for one gradient-descent epoch for
# fitting an MLE estimate of logistic regression with gradient descent (should
# return a tuple of two functions; see optimize)
def gd(y, X):
    def objective(w):
        # YOUR CODE HERE
        return -l(y, X, w)

    def update(w, eps):
        # YOUR CODE HERE
        grad = dl(y, X, w)
        return w + eps * grad

    return (objective, update)
```

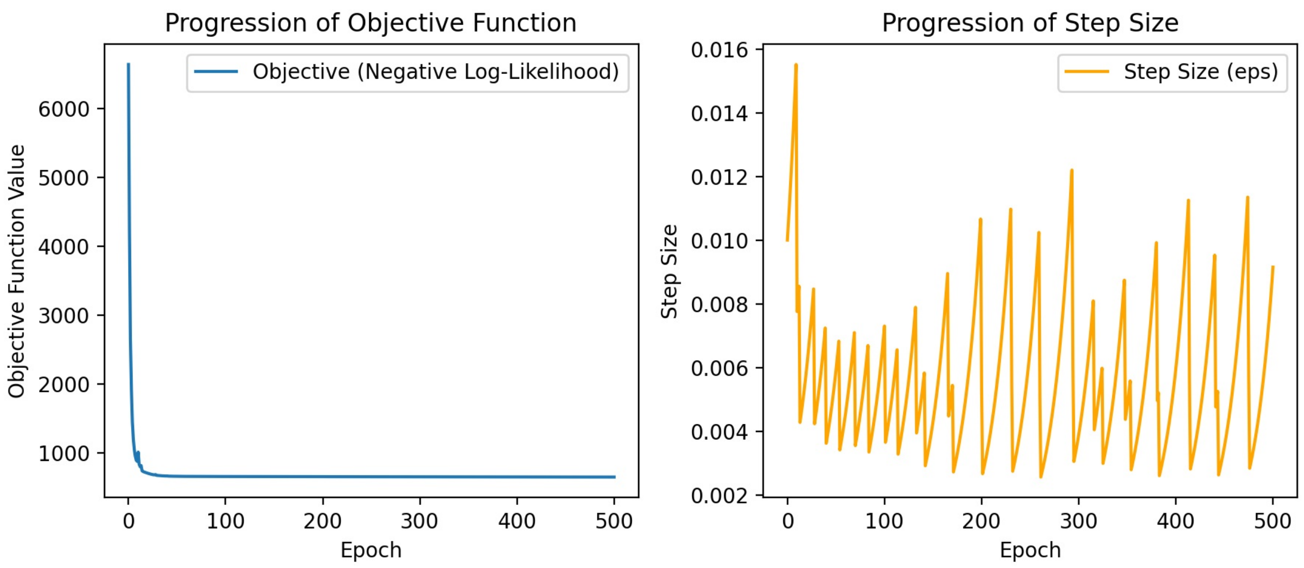
In [28]:
```python
# this should give
# [47066.641667825766,
#  array([  4.13777838e+01,  -1.56745627e+01,   5.75882538e+01,
#           1.14225143e+01,   5.54249703e+01,   5.99229049e+01,
#           7.11220141e+01,   4.84761728e+01,   5.78067289e+01,
#           4.54794720e+01,   7.14638492e+01,   1.51369386e+01,
#           3.36375739e+01,   2.15061217e+01,   5.78014255e+01,
#           6.72743066e+01,   7.00829312e+01,   5.29328088e+01,
```

```
#           6.16042473e+01,   5.50018510e+01,   8.94624817e+01,
#           2.74784480e+01,   8.51763599e+01,   5.60363965e+01,
#          -2.55865589e+01,  -1.53788213e+01,  -4.67015412e+01,
#          -2.50356570e+00,  -3.85357592e+00,  -2.21819155e+00,
#           3.32098671e+00,   3.86933390e+00,  -2.00309898e+01,
#           3.84684492e+00,  -2.19847927e-01,  -1.29775457e+00,
#          -1.28374302e+01,  -2.78303173e+00,  -5.61671182e+00,
#           1.73657121e+01,  -6.81197570e+00,  -1.20249002e+01,
#           2.65789491e+00,  -1.39557852e+01,  -2.01135653e+01,
#          -2.72134051e+01,  -9.45952961e-01,  -1.02239111e+01,
#           1.52794293e-04,  -5.18938123e-01,  -3.19717561e+00,
#           4.62953437e+01,   7.87893022e+01,   1.88618651e+01,
#           2.85195027e+01,   5.04698358e+01,   6.41240689e+01])
f, update = gd(y, Xz)
[f(np.linspace(-5, 5, D)), update(np.linspace(-5, -5, D), 0.1)]
```

Out[28]:
```
[47066.641667825774,
 array([ 4.13777838e+01, -1.56745627e+01,  5.75882538e+01,  1.14225143e+01,
         5.54249703e+01,  5.99229049e+01,  7.11220141e+01,  4.84761728e+01,
         5.78067289e+01,  4.54794720e+01,  7.14638492e+01,  1.51369386e+01,
         3.36375739e+01,  2.15061217e+01,  5.78014255e+01,  6.72743066e+01,
         7.00829312e+01,  5.29328088e+01,  6.16042473e+01,  5.50018510e+01,
         8.94624817e+01,  2.74784480e+01,  8.51763599e+01,  5.60363965e+01,
        -2.55865589e+01, -1.53788213e+01, -4.67015412e+01, -2.50356570e+00,
        -3.85357592e+00, -2.21819155e+00,  3.32098671e+00,  3.86933390e+00,
        -2.00309898e+01,  3.84684492e+00, -2.19847927e-01, -1.29775457e+00,
        -1.28374302e+01, -2.78303173e+00, -5.61671182e+00,  1.73657121e+01,
        -6.81197570e+00, -1.20249002e+01,  2.65789491e+00, -1.39557852e+01,
        -2.01135653e+01, -2.72134051e+01, -9.45952961e-01, -1.02239111e+01,
         1.52794293e-04, -5.18938123e-01, -3.19717561e+00,  4.62953437e+01,
         7.87893022e+01,  1.88618651e+01,  2.85195027e+01,  5.04698358e+01,
         6.41240689e+01])]
```

In [29]:
```
# you can run gradient descent!
numpy.random.seed(0)
w0 = np.random.normal(size=D)
wz_gd, vz_gd, ez_gd = optimize(gd(y, Xz), w0, nepochs=500, verbose=False)
```

In [30]:
```
# look at how gradient descent made progess
# YOUR CODE HERE
# Plot the progression of the objective function value (negative log-likelihood)
plt.figure(figsize=(9, 4))

plt.subplot(1, 2, 1)
plt.plot(vz_gd, label="Objective (Negative Log-Likelihood)")
plt.xlabel("Epoch")
plt.ylabel("Objective Function Value")
plt.title("Progression of Objective Function")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(ez_gd, label="Step Size (eps)", color="orange")
plt.xlabel("Epoch")
plt.ylabel("Step Size")
plt.title("Progression of Step Size")
plt.legend()

plt.tight_layout()
plt.show()
```



2d Stochastic gradient descent

In [31]:
```python
def sgdepoch(y, X, w, eps):
    """Run one SGD epoch and return the updated weight vector. """
    # Run N stochastic gradient steps (without replacement). Do not rescale each
    # step by factor N (i.e., proceed differently than in the lecture slides).
    # YOUR CODE HERE
    indices = np.random.permutation(len(y))

    for i in indices:
        xi = X[i]
        yi = y[i]

        gradient = (yi - sigma(xi @ w)) * xi
        w += eps * gradient

    return w
```

In [32]:
```python
# when you run this multiple times, with 50% probability you should get the
# following result (there is one other result which is very close):
# array([ -3.43689655e+02,  -1.71161311e+02,  -5.71093536e+02,
#         -5.16478220e+01,   4.66294348e+02,  -3.71589878e+02,
#          5.21493183e+02,   1.25699230e+03,   8.33804130e+02,
#          5.63185399e+02,   1.32761302e+03,  -2.64104011e+02,
#          7.10693307e+02,  -1.75497331e+02,  -1.94174427e+02,
#          1.11641507e+02,  -3.30817509e+02,  -3.46754913e+02,
#          8.48722111e+02,  -1.89136304e+02,  -4.25693844e+02,
#         -1.23084189e+02,  -2.95894797e+02,  -2.35789333e+02,
#         -3.38695243e+02,  -3.05642830e+02,  -2.28975383e+02,
#         -2.38075137e+02,  -1.66702530e+02,  -2.27341599e+02,
#         -1.77575620e+02,  -1.49093855e+02,  -1.70028859e+02,
#         -1.50243833e+02,  -1.82986008e+02,  -2.41143708e+02,
#         -3.31047159e+02,  -5.79991185e+01,  -1.98477863e+02,
#         -1.91264948e+02,  -1.17371919e+02,  -1.66953779e+02,
#         -2.01472565e+02,  -1.23330949e+02,  -3.00857740e+02,
#         -1.95853348e+02,  -7.44868073e+01,  -1.11172370e+02,
#         -1.57618226e+02,  -1.25729512e+00,  -1.45536466e+02,
#         -1.43362438e+02,  -3.00429708e+02,  -9.84391082e+01,
#         -4.54152047e+01,  -5.26492232e+01,  -1.45175427e+02])
sgdepoch(y[1:3], Xz[1:3, :], np.linspace(-5, 5, D), 1000)
```

Out[32]:
```
array([-3.43689655e+02, -1.71161311e+02, -5.71093536e+02, -5.16478220e+01,
        4.66294348e+02, -3.71589878e+02,  5.21493183e+02,  1.25699230e+03,
        8.33804130e+02,  5.63185399e+02,  1.32761302e+03, -2.64104011e+02,
        7.10693307e+02, -1.75497331e+02, -1.94174427e+02,  1.11641507e+02,
       -3.30817509e+02, -3.46754913e+02,  8.48722111e+02, -1.89136304e+02,
       -4.25693844e+02, -1.23084189e+02, -2.95894797e+02, -2.35789333e+02,
       -3.38695243e+02, -3.05642830e+02, -2.28975383e+02, -2.38075137e+02,
       -1.66702530e+02, -2.27341599e+02, -1.77575620e+02, -1.49093855e+02,
       -1.70028859e+02, -1.50243833e+02, -1.82986008e+02, -2.41143708e+02,
       -3.31047159e+02, -5.79991185e+01, -1.98477863e+02, -1.91264948e+02,
       -1.17371919e+02, -1.66953779e+02, -2.01472565e+02, -1.23330949e+02,
       -3.00857740e+02, -1.95853348e+02, -7.44868073e+01, -1.11172370e+02,
       -1.57618226e+02, -1.25729512e+00, -1.45536466e+02, -1.43362438e+02,
       -3.00429708e+02, -9.84391082e+01, -4.54152047e+01, -5.26492232e+01,
       -1.45175427e+02])
```

In [33]:
```python
# define the objective and update function for one gradient-descent epoch for
# fitting an MLE estimate of logistic regression with stochastic gradient descent
# (should return a tuple of two functions; see optimize)
def sgd(y, X):
    def objective(w):
        # YOUR CODE HERE
        return -l(y, X, w)

    def update(w, eps):
        return sgdepoch(y, X, w, eps)

    return (objective, update)
```

In [34]:
```python
# with 50% probability, you should get:
# [40.864973045695081,
#  array([ -3.43689655e+02,  -1.71161311e+02,  -5.71093536e+02,
#          -5.16478220e+01,   4.66294348e+02,  -3.71589878e+02,
#           5.21493183e+02,   1.25699230e+03,   8.33804130e+02,
#           5.63185399e+02,   1.32761302e+03,  -2.64104011e+02,
#           7.10693307e+02,  -1.75497331e+02,  -1.94174427e+02,
#           1.11641507e+02,  -3.30817509e+02,  -3.46754913e+02,
#           8.48722111e+02,  -1.89136304e+02,  -4.25693844e+02,
#          -1.23084189e+02,  -2.95894797e+02,  -2.35789333e+02,
#          -3.38695243e+02,  -3.05642830e+02,  -2.28975383e+02,
#          -2.38075137e+02,  -1.66702530e+02,  -2.27341599e+02,
```

```
#          -1.77575620e+02,  -1.49093855e+02,  -1.70028859e+02,
#          -1.50243833e+02,  -1.82986008e+02,  -2.41143708e+02,
#          -3.31047159e+02,  -5.79991185e+01,  -1.98477863e+02,
#          -1.91264948e+02,  -1.17371919e+02,  -1.66953779e+02,
#          -2.01472565e+02,  -1.23330949e+02,  -3.00857740e+02,
#          -1.95853348e+02,  -7.44868073e+01,  -1.11172370e+02,
#          -1.57618226e+02,  -1.25729512e+00,  -1.45536466e+02,
#          -1.43362438e+02,  -3.00429708e+02,  -9.84391082e+01,
#          -4.54152047e+01,  -5.26492232e+01,  -1.45175427e+02])]
f, update = sgd(y[1:3], Xz[1:3, :])
[f(np.linspace(-5, 5, D)), update(np.linspace(-5, 5, D), 1000)]
```

Out[34]:

```
[40.86497304569509,
 array([-3.43689655e+02, -1.71161311e+02, -5.71093536e+02, -5.16478220e+01,
         4.66294348e+02, -3.71589878e+02,  5.21493183e+02,  1.25699230e+03,
         8.33804130e+02,  5.63185399e+02,  1.32761302e+03, -2.64104011e+02,
         7.10693307e+02, -1.75497331e+02, -1.94174427e+02,  1.11641507e+02,
        -3.30817509e+02, -3.46754913e+02,  8.48722111e+02, -1.89136304e+02,
        -4.25693844e+02, -1.23084189e+02, -2.95894797e+02, -2.35789333e+02,
        -3.38695243e+02, -3.05642830e+02, -2.28975383e+02, -2.38075137e+02,
        -1.66702530e+02, -2.27341599e+02, -1.77575620e+02, -1.49093855e+02,
        -1.70028859e+02, -1.50243833e+02, -1.82986008e+02, -2.41143708e+02,
        -3.31047159e+02, -5.79991185e+01, -1.98477863e+02, -1.91264948e+02,
        -1.17371919e+02, -1.66953779e+02, -2.01472565e+02, -1.23330949e+02,
        -3.00857740e+02, -1.95853348e+02, -7.44868073e+01, -1.11172370e+02,
        -1.57618226e+02, -1.25729512e+00, -1.45536466e+02, -1.43362438e+02,
        -3.00429708e+02, -9.84391082e+01, -4.54152047e+01, -5.26492232e+01,
        -1.45175427e+02])]
```

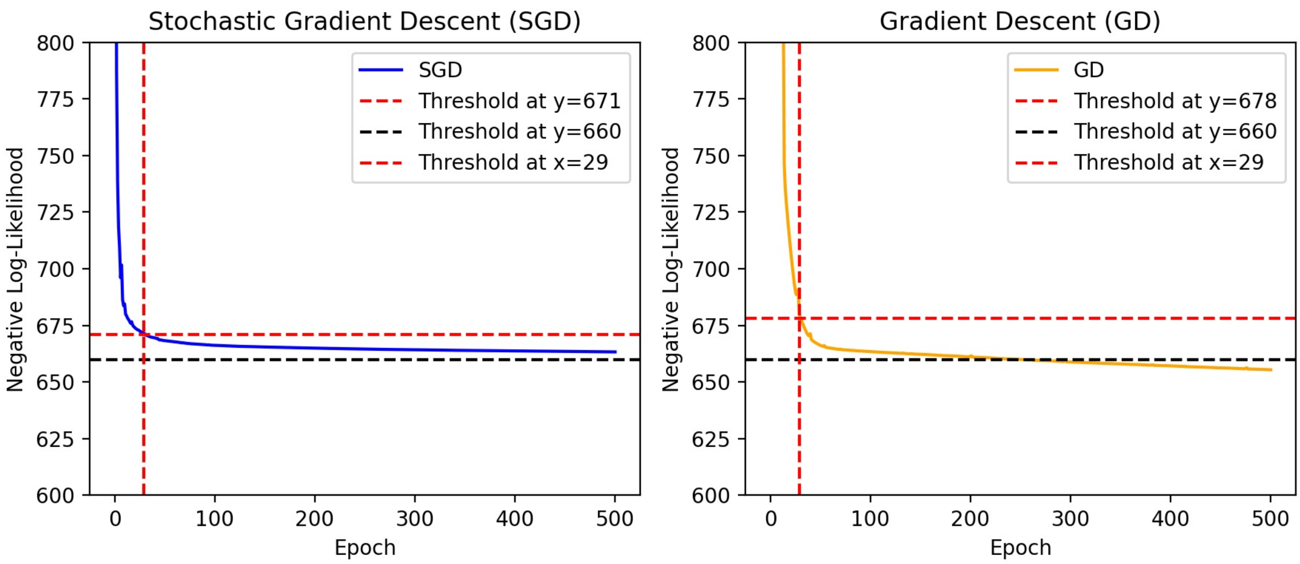
In [35]:

```
# you can run stochastic gradient descent!
wz_sgd, vz_sgd, ez_sgd = optimize(sgd(y, Xz), w0, nepochs=500, verbose=False)
```

2e Compare GD and SGD

In [36]:

```
# YOUR CODE HERE
np.random.seed(0)

plt.figure(figsize=(9, 4))

plt.subplot(1, 2, 1)
plt.plot(vz_sgd, label='SGD', color='blue')
plt.title('Stochastic Gradient Descent (SGD)')
plt.xlabel('Epoch')
plt.ylabel('Negative Log-Likelihood')
plt.ylim(600, 800)
plt.axhline(y=671, color='red', linestyle='--', label='Threshold at y=671')
plt.axhline(y=660, color='black', linestyle='--', label='Threshold at y=660')
plt.axvline(x=29, color='red', linestyle='--', label='Threshold at x=29')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(vz_gd, label='GD', color='orange')
plt.title('Gradient Descent (GD)')
plt.xlabel('Epoch')
plt.ylabel('Negative Log-Likelihood')
plt.ylim(600, 800)
plt.axhline(y=678, color='red', linestyle='--', label='Threshold at y=678')
plt.axhline(y=660, color='black', linestyle='--', label='Threshold at y=660')
plt.axvline(x=29, color='red', linestyle='--', label='Threshold at x=29')
plt.legend()

plt.tight_layout()
plt.show()
```



In [37]:
```
print('1) SVD min loglikelihood: ', np.min(vz_sgd), '\n2) GD min loglikelihood : ' , np.min(vz_gd))
print('*****************************************')
print('1) SVD mean loglikelihood: ', np.mean(vz_sgd), '\n2) GD mean loglikelihood : ' , np.mean(vz_gd))
print('*****************************************')
print('1) SVD median loglikelihood: ', np.median(vz_sgd), '\n2) GD median loglikelihood : ' , np.median(vz_gd))
print('*****************************************')
print('1) SVD 1st quartile loglikelihood: ', np.percentile(vz_sgd, 25), '\n2) GD 1st quartile loglikelihood : '
print('*****************************************')
print('1) SVD 3rd quartile loglikelihood: ', np.percentile(vz_sgd, 75), '\n2) GD 3rd quartile loglikelihood : '
```

```
1) SVD min loglikelihood:  663.3023109198814
2) GD min loglikelihood :  655.413496469943
*****************************************
1) SVD mean loglikelihood:  678.626118735844
2) GD mean loglikelihood :  693.8859677697319
*****************************************
1) SVD median loglikelihood:  664.5794412166027
2) GD median loglikelihood :  659.9852259808722
*****************************************
1) SVD 1st quartile loglikelihood:  663.8686408677954
2) GD 1st quartile loglikelihood :  657.5670859999374
*****************************************
1) SVD 3rd quartile loglikelihood:  665.7772597882638
2) GD 3rd quartile loglikelihood :  662.8279655964287
```

3 Prediction

In [38]:
```
def predict(Xtest, w):
    """Returns vector of predicted confidence values for logistic regression with
weight vector w."""
    # YOUR CODE HERE
    return sigma(Xtest @ w)

def classify(Xtest, w):
    """Returns 0/1 vector of predicted class labels for logistic regression with
weight vector w."""
    # YOUR CODE HERE
    return np.where(predict(Xtest, w) > 0.5, 1, 0)
```

In [39]:
```
yhat_sgd = predict(Xtestz, wz_sgd)
ypred_sgd = classify(Xtestz, wz_sgd)
print(sklearn.metrics.confusion_matrix(ytest, ypred_sgd))  # true x predicted
```

```
[[886  55]
 [ 72 523]]
```

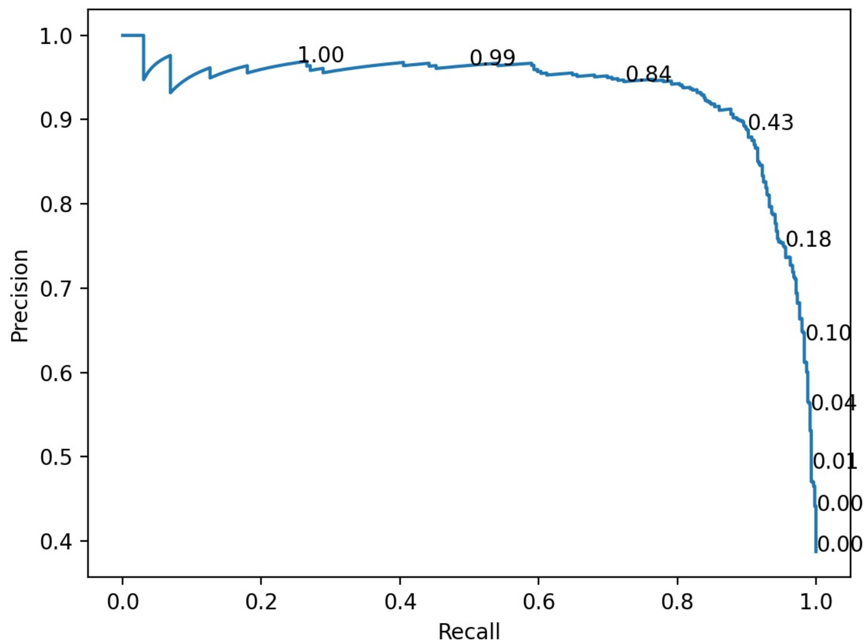
In [40]:
```
# Example: confusion matrix
yhat = predict(Xtestz, wz_gd)
ypred = classify(Xtestz, wz_gd)
print(sklearn.metrics.confusion_matrix(ytest, ypred))  # true x predicted
```

```
[[887  54]
 [ 71 524]]
```

In [41]:
```
# Example: classification report
print('GD:')
print(sklearn.metrics.classification_report(ytest, ypred))
print('************************************')
print('SGD: ')
print(sklearn.metrics.classification_report(ytest, ypred_sgd))
```

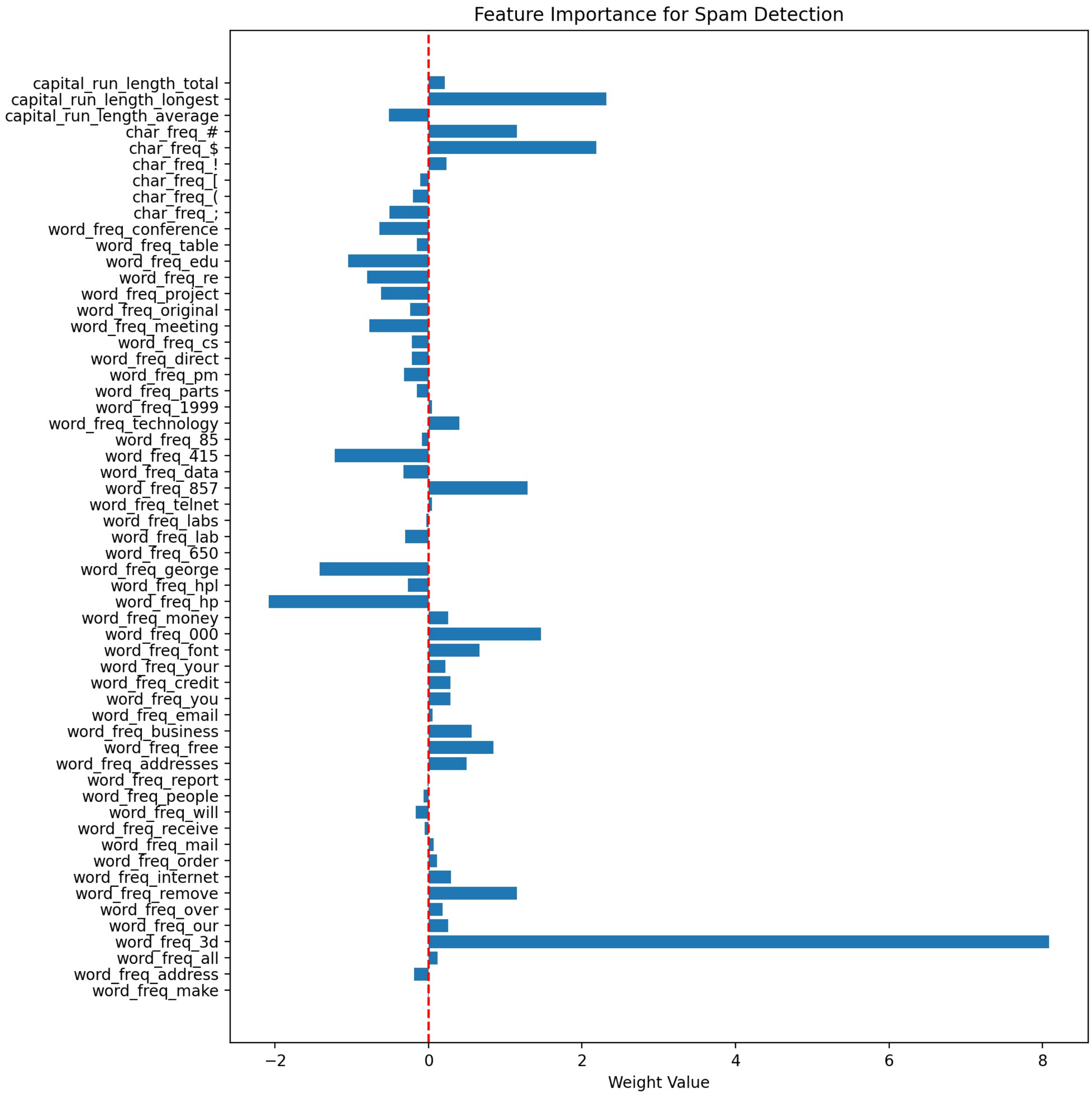
```
GD:
              precision    recall  f1-score   support

           0       0.93      0.94      0.93       941
           1       0.91      0.88      0.89       595

    accuracy                           0.92      1536
   macro avg       0.92      0.91      0.91      1536
weighted avg       0.92      0.92      0.92      1536

************************************
SGD:
              precision    recall  f1-score   support

           0       0.92      0.94      0.93       941
           1       0.90      0.88      0.89       595

    accuracy                           0.92      1536
   macro avg       0.91      0.91      0.91      1536
weighted avg       0.92      0.92      0.92      1536
```

In [42]:
```
# Example: precision-recall curve (with annotated thresholds)
nextplot()
precision, recall, thresholds = sklearn.metrics.precision_recall_curve(ytest, yhat)
plt.plot(recall, precision)
for x in np.linspace(0, 1, 10, endpoint=False):
    index = int(x * (precision.size - 1))
    plt.text(recall[index], precision[index], "{:3.2f}".format(thresholds[index]))
plt.xlabel("Recall")
plt.ylabel("Precision")
```



Out[42]: Text(0, 0.5, 'Precision')

In [43]:
```python
# Explore which features are considered important
# YOUR CODE HERE
def plot_feature_importance(w, feature_names):
    plt.figure(figsize=(10, 10))
    plt.barh(feature_names, w)
    plt.xlabel('Weight Value')
    plt.title('Feature Importance for Spam Detection')
    plt.axvline(0, color='red', linestyle='--')
    plt.tight_layout()
    plt.show()

plot_feature_importance(wz_gd, features)
```



4 Maximum Aposteriori Estimation

4a Gradient Descent

In [44]:
```python
def l_l2(y, X, w, lambda_):
    """Log-density of posterior of logistic regression with weights w and L2
regularization parameter lambda_"""
    # YOUR CODE HERE
    log_likelihood = l(y, X, w)
    log_prior = -0.5 * lambda_ * np.sum(w ** 2)
    return log_likelihood + log_prior
```

In [45]:
```python
# this should give:
# [-47066.641667825766, -47312.623810682911]
[l_l2(y, Xz, np.linspace(-5, 5, D), 0), l_l2(y, Xz, np.linspace(-5, 5, D), 1)]
```

Out[45]: [-47066.641667825774, -47312.62381068292]

In [46]:
```python
def dl_l2(y, X, w, lambda_):
    """Gradient of log-density of posterior of logistic regression with weights w
and L2 regularization parameter lambda_."""
    # YOUR CODE HERE
    gradient_likelihood = dl(y, X, w)
    gradient_prior = -lambda_ * w  # Derivative of the Gaussian prior
    return gradient_likelihood + gradient_prior
```

In [47]:
```python
# this should give:
# [array([  551.33985842,   143.84116318,   841.83373606,   156.87237578,
#           802.61217579,   795.96202907,   920.69045803,   621.96516752,
#           659.18724769,   470.81259805,   771.32406968,   352.40325626,
#           455.66972482,   234.36600888,   562.45454038,   864.83981264,
#           787.19723703,   649.48042176,   902.6478154 ,   544.00539886,
#          1174.78638035,   120.3598967 ,   839.61141672,   633.30453444,
#          -706.66815087,  -630.2039816 ,  -569.3451386 ,  -527.50996698,
#          -359.53701083,  -476.64334832,  -411.60620464,  -375.11950586,
#          -345.37195689,  -376.22044258,  -407.31761977,  -456.23251936,
#          -596.86960184,  -107.97072355,  -394.82170044,  -229.18125598,
#          -288.46356547,  -362.13402385,  -450.87896465,  -277.03932676,
#          -414.99293368,  -452.28771693,  -167.54649092,  -270.9043748 ,
#          -252.20140951,  -357.72497343,  -259.12468742,   418.35938483,
#           604.54173228,    43.10390907,   152.24258478,   378.16731033,
#           416.12032881]),
#  array([  556.33985842,   148.66259175,   846.4765932 ,   161.33666149,
#           806.89789007,   800.06917193,   924.61902946,   625.71516752,
#           662.75867626,   474.20545519,   774.5383554 ,   355.43897054,
#           458.52686767,   237.04458031,   564.95454038,   867.16124121,
#           789.34009417,   651.44470748,   904.43352968,   545.61254171,
#          1176.21495178,   121.6098967 ,   840.68284529,   634.19739158,
#          -705.95386516,  -629.66826731,  -568.98799574,  -527.33139555,
#          -359.53701083,  -476.82191975,  -411.9633475 ,  -375.65522015,
#          -346.08624261,  -377.11329972,  -408.38904835,  -457.48251936,
#          -598.29817327,  -109.57786641,  -396.60741472,  -231.14554169,
#          -290.60642261,  -364.45545242,  -453.37896465,  -279.71789819,
#          -417.85007654,  -455.32343122,  -170.76077664,  -274.29723194,
#          -255.77283808,  -361.47497343,  -263.05325885,   414.25224198,
#           600.25601799,    38.63962335,   147.59972763,   373.34588176,
#           411.12032881])]
[dl_l2(y, Xz, np.linspace(-5, 5, D), 0), dl_l2(y, Xz, np.linspace(-5, 5, D), 1)]
```

Out[47]:
```
[array([ 551.33985842,  143.84116318,  841.83373606,  156.87237578,
         802.61217579,  795.96202907,  920.69045803,  621.96516752,
         659.18724769,  470.81259805,  771.32406968,  352.40325626,
         455.66972482,  234.36600888,  562.45454038,  864.83981264,
         787.19723703,  649.48042176,  902.6478154 ,  544.00539886,
        1174.78638035,  120.3598967 ,  839.61141672,  633.30453444,
        -706.66815087, -630.2039816 , -569.3451386 , -527.50996698,
        -359.53701083, -476.64334832, -411.60620464, -375.11950586,
        -345.37195689, -376.22044258, -407.31761977, -456.23251936,
        -596.86960184, -107.97072355, -394.82170044, -229.18125598,
        -288.46356547, -362.13402385, -450.87896465, -277.03932676,
        -414.99293368, -452.28771693, -167.54649092, -270.9043748 ,
        -252.20140951, -357.72497343, -259.12468742,  418.35938483,
         604.54173228,   43.10390907,  152.24258478,  378.16731033,
         416.12032881]),
 array([ 556.33985842,  148.66259175,  846.4765932 ,  161.33666149,
         806.89789007,  800.06917193,  924.61902946,  625.71516752,
         662.75867626,  474.20545519,  774.5383554 ,  355.43897054,
         458.52686767,  237.04458031,  564.95454038,  867.16124121,
         789.34009417,  651.44470748,  904.43352968,  545.61254171,
        1176.21495178,  121.6098967 ,  840.68284529,  634.19739158,
        -705.95386516, -629.66826731, -568.98799574, -527.33139555,
        -359.53701083, -476.82191975, -411.9633475 , -375.65522015,
        -346.08624261, -377.11329972, -408.38904835, -457.48251936,
        -598.29817327, -109.57786641, -396.60741472, -231.14554169,
        -290.60642261, -364.45545242, -453.37896465, -279.71789819,
        -417.85007654, -455.32343122, -170.76077664, -274.29723194,
        -255.77283808, -361.47497343, -263.05325885,  414.25224198,
         600.25601799,   38.63962335,  147.59972763,  373.34588176,
         411.12032881])]
```

In [48]:
```python
# now define the (f,update) tuple for optimize for logistic regression, L2
# regularization, and gradient descent
def gd_l2(y, X, lambda_):
    # YOUR CODE HERE
    def objective(w):
        return -l_l2(y, X, w, lambda_)

    def update(w, eps):
```

```
        grad = dl_l2(y, X, w, lambda_)
        return w + eps * grad

    return (objective, update)
```

In [49]:
```
# let's run!
lambda_ = 100
wz_gd_l2, vz_gd_l2, ez_gd_l2 = optimize(gd_l2(y, Xz, lambda_), w0, nepochs=500, verbose=False)
```

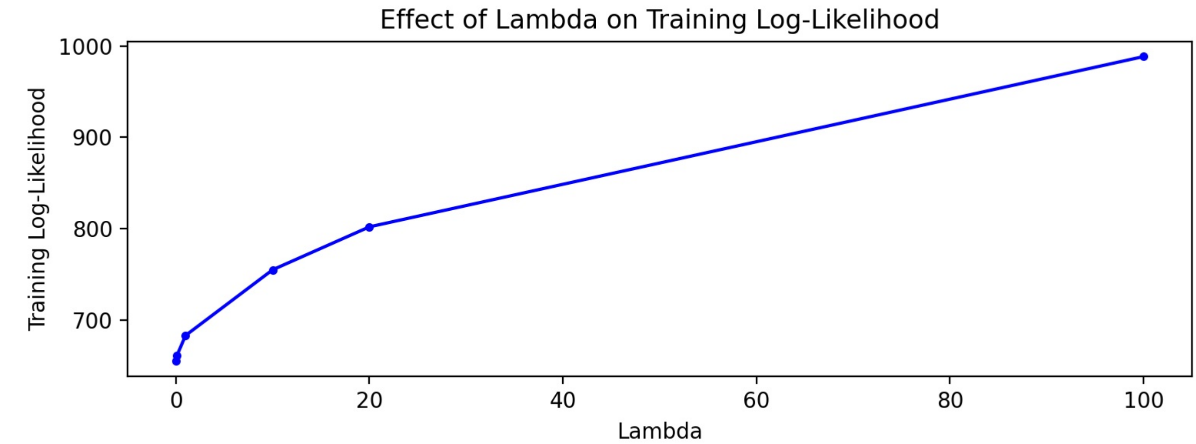
4b Effect of Prior

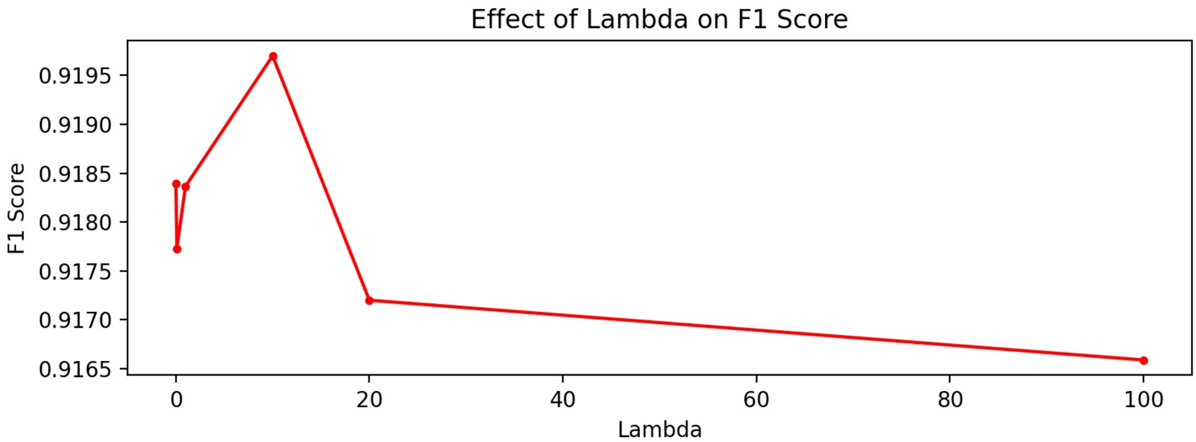
In [50]:
```
# YOUR CODE HERE
# Define the range of lambda values to test
lambda_values = [0.01, 0.1, 1, 10, 20, 100]
results = {}

for lambda_ in lambda_values:
    wz_gd_l2, vz_gd_l2, ez_gd_l2 = optimize(gd_l2(y, Xz, lambda_), w0, nepochs=500, verbose=False)

    yhat_l2 = predict(Xtestz, wz_gd_l2)
    ypred_l2 = classify(Xtestz, wz_gd_l2)

    confusion_matrix = sklearn.metrics.confusion_matrix(ytest, ypred_l2)
    classification_report = sklearn.metrics.classification_report(ytest, ypred_l2, output_dict=True)

    results[lambda_] = {
        "Confusion Matrix": confusion_matrix,
        "Classification Report": classification_report,
        "Minimum Log-Likelihood (Training)": vz_gd_l2[-1],
        "Test Accuracy": classification_report["accuracy"],
        "Test F1 Score": classification_report["weighted avg"]["f1-score"],
        "Minimum Log-Likelihood (Test)": -l(ytest, Xtestz, wz_gd_l2)
    }

for lambda_, metrics in results.items():
    print(f"\nLambda: {lambda_}")
    print("Minimum Log-Likelihood (Training): ", metrics["Minimum Log-Likelihood (Training)"])
    print("Test Accuracy:", metrics["Test Accuracy"])
    print("Test F1:", metrics["Test F1 Score"])
    print("Minimum Log-Likelihood (Test):", metrics["Minimum Log-Likelihood (Test)"])
```

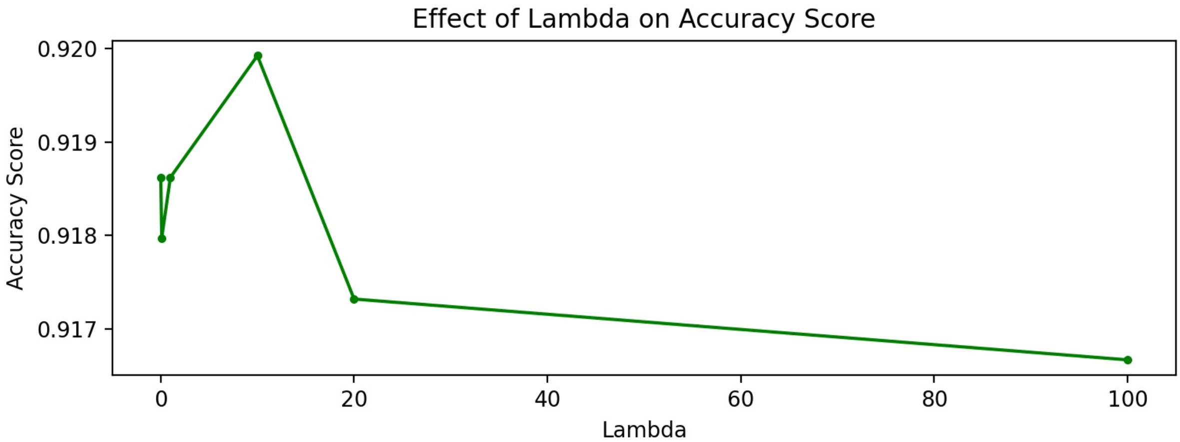
```
Lambda: 0.01
Minimum Log-Likelihood (Training):  654.8433538081598
Test Accuracy: 0.9186197916666666
Test F1: 0.9183943410814998
Minimum Log-Likelihood (Test): 426.4721356148674

Lambda: 0.1
Minimum Log-Likelihood (Training):  660.6152784238415
Test Accuracy: 0.91796875
Test F1: 0.9177273862717801
Minimum Log-Likelihood (Test): 427.3115437654666

Lambda: 1
Minimum Log-Likelihood (Training):  682.8501567926841
Test Accuracy: 0.9186197916666666
Test F1: 0.9183662634796539
Minimum Log-Likelihood (Test): 432.93634622996404

Lambda: 10
Minimum Log-Likelihood (Training):  754.8524204027971
Test Accuracy: 0.919921875
Test F1: 0.9197000316241958
Minimum Log-Likelihood (Test): 436.3227045757097

Lambda: 20
Minimum Log-Likelihood (Training):  801.9128109439284
Test Accuracy: 0.9173177083333334
Test F1: 0.9171994199922072
Minimum Log-Likelihood (Test): 440.7307420926481

Lambda: 100
Minimum Log-Likelihood (Training):  988.511839602703
Test Accuracy: 0.9166666666666666
Test F1: 0.9165879332722552
Minimum Log-Likelihood (Test): 476.74076719221745
```

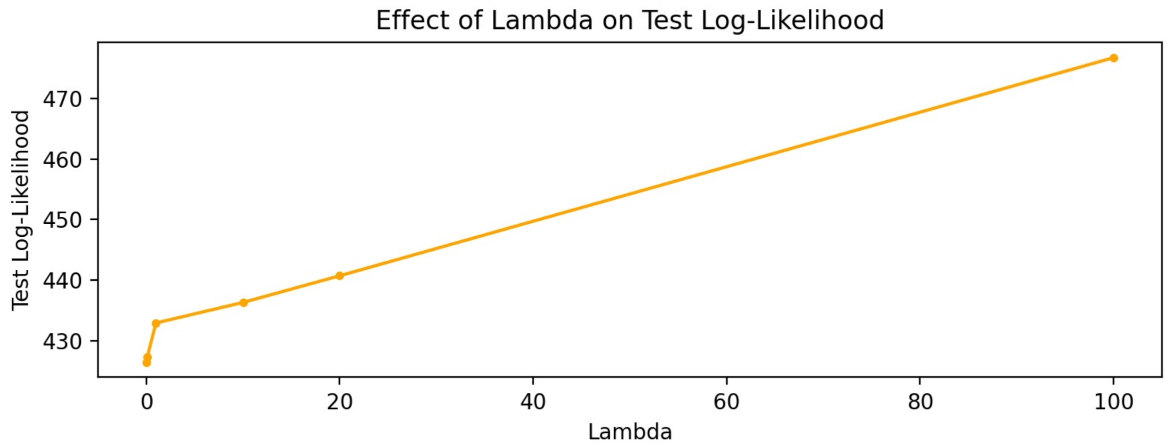
In [51]:
```
lambda_vals = list(results.keys())
min_log_likelihoods = [metrics["Minimum Log-Likelihood (Training)"] for metrics in results.values()]
test_accuracies = [metrics["Test Accuracy"] for metrics in results.values()]
f1_scores = [metrics["Test F1 Score"] for metrics in results.values()]
test_min_log_likelihoods = [metrics["Minimum Log-Likelihood (Test)"] for metrics in results.values()]
```

```python
plt.figure(figsize=(8, 12))

plt.subplot(4, 1, 1)
plt.plot(lambda_vals, min_log_likelihoods, marker='o', color='blue', markersize=3)
plt.xlabel("Lambda")
plt.ylabel("Training Log-Likelihood")
plt.title("Effect of Lambda on Training Log-Likelihood")

plt.subplot(4, 1, 2)
plt.plot(lambda_vals, f1_scores, marker='o', color='red', markersize=3)
plt.xlabel("Lambda")
plt.ylabel("F1 Score")
plt.title("Effect of Lambda on F1 Score")

plt.subplot(4, 1, 3)
plt.plot(lambda_vals, test_accuracies, marker='o', color='green', markersize=3)
plt.xlabel("Lambda")
plt.ylabel("Accuracy Score")
plt.title("Effect of Lambda on Accuracy Score")

plt.subplot(4, 1, 4)
plt.plot(lambda_vals, test_min_log_likelihoods, marker='o', color='orange', markersize=3)
plt.xlabel("Lambda")
plt.ylabel("Test Log-Likelihood")
plt.title("Effect of Lambda on Test Log-Likelihood")

plt.tight_layout()
plt.show()
```







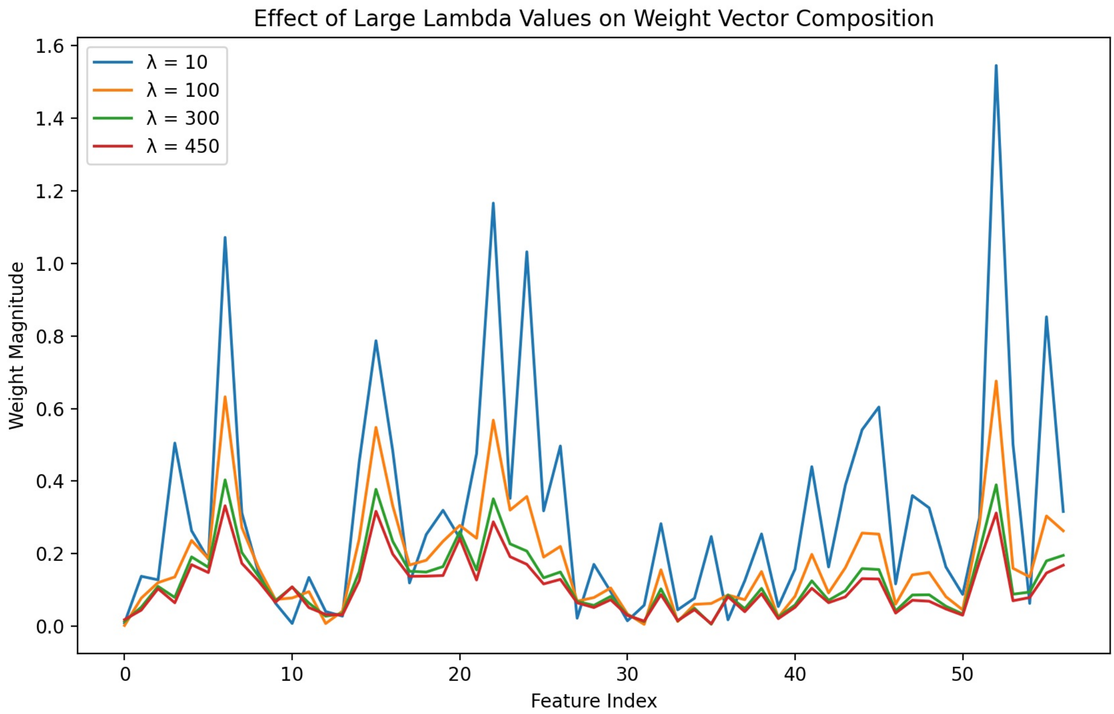


4c Composition of Weight Vector

In [52]:

```
# YOUR CODE HERE
large_lambda_values = [10, 100, 300, 450]
lambda_weights = {}

for lambda_ in large_lambda_values:
    wz_gd_l2, _, _ = optimize(gd_l2(y, Xz, lambda_), w0, nepochs=500, verbose=False)
    lambda_weights[lambda_] = wz_gd_l2
```

```python
plt.figure(figsize=(10, 6))
for lambda_, weights in lambda_weights.items():
    plt.plot(np.abs(weights), label=f'λ = {lambda_}')
plt.xlabel("Feature Index")
plt.ylabel("Weight Magnitude")
plt.title("Effect of Large Lambda Values on Weight Vector Composition")
plt.legend()
plt.show()
```



In [53]:
```python
features[np.argsort(lambda_weights[450])[::-1][:5]].tolist()
```

Out[53]:
```
['word_freq_remove',
 'word_freq_free',
 'char_freq_$',
 'word_freq_000',
 'word_freq_your']
```

5 Exploration (optional)

Try gradient descent on the original data without using z-scores.

In [54]:
```python
numpy.random.seed(0)
w0 = np.random.normal(size=D)
wz_gd_uns, vz_gd_uns, ez_gd_uns = optimize(gd(y, X), w0, nepochs=500, verbose=False)
```

```
/var/folders/0s/63ktljj55t12qgn_8z4nq4wm0000gn/T/ipykernel_38343/4219900393.py:7: RuntimeWarning: overflow encou
ntered in exp
  return -np.log1p(np.exp(-x))
/var/folders/0s/63ktljj55t12qgn_8z4nq4wm0000gn/T/ipykernel_38343/743502101.py:15: RuntimeWarning: invalid value
encountered in multiply
  log_likelihood = np.sum(y * logsigma(z) + (1 - y) * logsigma(-z))
/var/folders/0s/63ktljj55t12qgn_8z4nq4wm0000gn/T/ipykernel_38343/2533977827.py:5: RuntimeWarning: overflow encou
ntered in exp
  return 1 / (1 + np.exp(-x))
```

In [55]:
```python
print(vz_gd_uns[-10:])
```

```
[nan nan nan nan nan nan nan nan nan nan]
```

Add a bias feature (make sure that you do not scale it).

In [56]:
```python
def add_bias_feature(X):
    return np.hstack((np.ones((X.shape[0], 1)), X))
```

In [57]:
```python
np.random.seed(0)

Xz_bias = add_bias_feature(Xz)
D = Xz_bias.shape[1]
w0 = np.random.normal(size=D)
```

In [58]:
```
wz_gd, vz_gd, ez_gd = optimize(gd(y, Xz_bias), w0, nepochs=500, verbose=False)
print('min log likelihood: ', np.min(vz_gd))
```

```
min log likelihood:  590.8903562741727
```

In [59]:
```
Xtestz_bias = add_bias_feature(Xtestz)
```

In [60]:
```
yhat_bias = predict(Xtestz_bias, wz_gd)
ypred_bias = classify(Xtestz_bias, wz_gd)
print(sklearn.metrics.confusion_matrix(ytest, ypred_bias))
print(sklearn.metrics.classification_report(ytest, ypred_bias))
```

```
[[895  46]
 [ 65 530]]
              precision    recall  f1-score   support

           0       0.93      0.95      0.94       941
           1       0.92      0.89      0.91       595

    accuracy                           0.93      1536
   macro avg       0.93      0.92      0.92      1536
weighted avg       0.93      0.93      0.93      1536
```

Try to reduce the training set size and compare MLE and MAP estimation (ideally using cross-validation).

In [61]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, f1_score
```

In [62]:
```
np.random.seed(0)

sample_size = int(0.5 * X.shape[0])
sample_indices = np.random.choice(range(0, sample_size), size=sample_size, replace=False)

kf = KFold(n_splits=10, shuffle=True, random_state=0)

mle_results, map_results = {'accuracy': [], 'f1': []}, {'accuracy': [], 'f1': []}

for train_index, val_index in kf.split(X[sample_indices,:]):
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]

    mean_train = np.mean(X_train, axis=0)
    std_train = np.std(X_train, axis=0)
    X_train_scaled = (X_train - mean_train) / std_train
    X_val_scaled = (X_val - mean_train) / std_train

    w0 = np.random.normal(size=X_train_scaled.shape[1])

    w_mle, v_mle, _ = optimize(gd(y_train, X_train_scaled), w0, nepochs=500, verbose=False)
    y_pred_mle = classify(X_val_scaled, w_mle)
    mle_results['accuracy'].append(accuracy_score(y_val, y_pred_mle))
    mle_results['f1'].append(f1_score(y_val, y_pred_mle, average='weighted'))

    lambda_ = 0.1
    w_map, v_map, _ = optimize(gd_l2(y_train, X_train_scaled, lambda_), w0, nepochs=500, verbose=False)
    y_pred_map = classify(X_val_scaled, w_map)
    map_results['accuracy'].append(accuracy_score(y_val, y_pred_map))
    map_results['f1'].append(f1_score(y_val, y_pred_map, average='weighted'))
```

In [63]:
```
for key in mle_results.keys():
    print(f'MLE -- CV {key} mean: {np.mean(mle_results[key]):.4f}')
print('******************')
for key in map_results.keys():
    print(f'MAP -- CV {key} mean: {np.mean(map_results[key]):.4f}')
```

```
MLE -- CV accuracy mean: 0.9452
MLE -- CV f1 mean: 0.9445
******************
MAP -- CV accuracy mean: 0.9445
MAP -- CV f1 mean: 0.9438
```

Run a logistic regression method from some existing library. Do you get the same results?

In [64]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
```

In [65]:
```
log_reg = LogisticRegression(penalty='l2', solver='lbfgs', max_iter=500, random_state=0)
log_reg.fit(Xz, y)
```

```
y_pred_test = log_reg.predict(Xtestz)

test_accuracy = accuracy_score(ytest, y_pred_test)
print("Test Accuracy:", test_accuracy)

print("Classification Report:")
print(classification_report(ytest, y_pred_test))
```

```
Test Accuracy: 0.9244791666666666
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.95      0.94       941
           1       0.92      0.88      0.90       595

    accuracy                           0.92      1536
   macro avg       0.92      0.92      0.92      1536
weighted avg       0.92      0.92      0.92      1536
```

5 Exploration: PyTorch

In [66]:

```
# if you want to experiment, here is an implementation of logistic
# regression in PyTorch
import math
import torch
import torch.nn as nn
import torch.utils.data
import torch.nn.functional as F

# prepare the data
Xztorch = torch.FloatTensor(Xz)
ytorch = torch.LongTensor(y)
train = torch.utils.data.TensorDataset(Xztorch, ytorch)


# manual implementation of logistic regression (without bias)
class LogisticRegression(nn.Module):
    def __init__(self, D, C):
        super(LogisticRegression, self).__init__()
        self.weights = torch.nn.Parameter(
            torch.randn(D, C) / math.sqrt(D)
        )  # xavier initialization
        self.register_parameter("W", self.weights)

    def forward(self, x):
        out = torch.matmul(x, self.weights)
        out = F.log_softmax(out)
        return out


# define the objective and update function. here we ignore the learning rates
# and parameters given to us by optimize (they are stored in the PyTorch model
# and optimizer, resp., instead)
def opt_pytorch():
    model = LogisticRegression(D, 2)
    criterion = nn.NLLLoss(reduction="sum")
    # change the next line to try different optimizers
    # optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    def objective(_):
        outputs = model(Xztorch)
        return criterion(outputs, ytorch)

    def update(_1, _2):
        for i, (examples, labels) in enumerate(train_loader):
            outputs = model(examples)
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        W = model.state_dict()["W"]
        w = W[:, 1] - W[:, 0]
        return w

    return (objective, update)
```

In [ ]:

```
# run the optimizer
learning_rate = 0.01
batch_size = 100  # number of data points to sample for gradient estimate
```

```
shuffle = True  # sample with replacement (false) or without replacement (true)

train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True)
wz_t, vz_t, _ = optimize(opt_pytorch(), None, nepochs=100, eps0=None, verbose=True)
```