housing-prices

October 6, 2025

```
[1]: # Continuation of House Prices from HW1
```

```
[16]: # Run this code to see markdowns easier
# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how to make markdown cells a different color in jupyter notebooks"
from IPython.display import Javascript

Javascript("""
var style = document.createElement('style');
style.innerHTML = `
  /* Target Markdown cells only */
  .jp-Notebook .jp-MarkdownOutput {
      background-color: #d8f0ff !important;
      border: none !important;           /* remove border */
      border-radius: 10px;
      padding: 12px !important;
      box-shadow: 0 0 8px rgba(112, 174, 224, 0.3);
  }
`;
document.head.appendChild(style);
""")
```

```
[16]: <IPython.core.display.Javascript object>
```

1 Continuation of Housing Prices

1.1 Feature Selection

In HW1, I used OLS with scikit-learn to test several models with MSE and R^2 scores. The final model ("Top 40") had MSE score of 9.035*10^8 and an R^2 score of 0.8567. The OLS by scikit-learn will be reperformed in this assignment, but this discussion is necessary since the preprocessing steps and decision to choose the "Top 40" feature set comes from HW1.

The preprocessing involved these following steps.

1) Impute missing values
2) Value encode high cardinality categorical values
3) One-hot encode other categorical values

4) Scale values between 0 and 1
5) Calculate mutual information scores and select top 50 features
6) Apply domain-relevant feature combinations and feature re-additions
7) Visualize and remove low correlation features
8) Evaluate models for top 10, 20, 30, 40, 50 mutual information features

[2]:
```
# Import
import pandas as pd
import torch

train_40 = pd.read_csv("./data-from-hw1/train_40.csv")
test_40 = pd.read_csv("./data-from-hw1/test_40.csv")
prices = pd.read_csv('./data-from-hw1/train.csv')['SalePrice']
print(train_40.shape)
```

```
(1460, 40)
```

1.2 OLS by gradient descent

I split the training set into training and validation sets.

[3]:
```
# Split training set into training and validation
from sklearn.model_selection import train_test_split

train_X, val_X, train_Y, val_Y = train_test_split(
    train_40, prices, test_size=0.3, random_state=42, shuffle=True
)
train_X.shape
train_X.head()
```

[3]:
```
      OverallQual  Neighborhood  GrLivArea  YearBuilt  TotalBsmtSF  \
135      0.666667      0.378134   0.253956   0.710145     0.213421
1452     0.444444      0.119470   0.139035   0.963768     0.089525
762      0.666667      0.532616   0.228523   0.992754     0.123732
932      0.888889      0.532616   0.295968   0.971014     0.311784
435      0.666667      0.416045   0.250000   0.898551     0.130769

      GarageArea  GarageCars  1stFlrSF  MSSubClass  ExterQual_TA  …  \
135     0.373766        0.50  0.309316    0.000000           1.0  …
1452    0.370240        0.50  0.169344    0.941176           1.0  …
762     0.433004        0.50  0.098669    0.235294           1.0  …
932     0.555712        0.75  0.360486    0.000000           0.0  …
435     0.387870        0.50  0.113125    0.235294           0.0  …

      BsmtUnfSF  BsmtQual_Ex  OverallCond  BsmtFinType1_GLQ  GarageFinish_Fin  \
135    0.558219          0.0        0.625               0.0               0.0
```

```
1452   0.000000          0.0        0.500              1.0              1.0
762    0.313356          0.0        0.500              1.0              0.0
932    0.815497          1.0        0.500              0.0              1.0
435    0.029966          0.0        0.625              1.0              0.0

      GarageType_Detchd  WoodDeckSF  Foundation_CBlock  HeatingQC_TA  YrSold
135                 0.0    0.114352                0.0           0.0    0.50
1452                0.0    0.000000                0.0           0.0    0.00
762                 0.0    0.197200                0.0           0.0    1.00
932                 0.0    0.000000                0.0           0.0    0.25
435                 0.0    0.184364                0.0           0.0    0.75

[5 rows x 40 columns]
```

[4]:
```
# Define model class as linear regression
# y = xW + b
W = torch.randn(40, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
train_X_tensor = torch.tensor(train_X.values, dtype = torch.float32)
train_Y_tensor = torch.tensor(train_Y.to_numpy().reshape(-1, 1), dtype=torch.
 ↪float32)
```

Then I defined training loops. In each loop, I use the mean squared error and an L2 regularization to define the loss function.

Thus, there are three hyperparameters: the learning rate, the regularization parameter, and the number of training loops.

[5]:
```
# Define one autodiff/update loop
def update_once(learning_rate, reg_param, W, b):
    prices_predicted = train_X_tensor @ W + b
    loss = torch.mean((prices_predicted - train_Y_tensor)**2) + (reg_param / 2)␣
 ↪* torch.sum(W**2)
    loss.backward()
    with torch.no_grad():  # disables autograd temporarily
        W.data -= learning_rate * W.grad
        b.data -= learning_rate * b.grad

    W.grad.zero_()
    b.grad.zero_()
    return loss.item(), W, b
```

[6]:
```
# Define how many updates
def update_N(learning_rate, reg_param, num_epochs, W, b):
    W = torch.randn(40, 1, requires_grad=True)
    b = torch.randn(1, requires_grad=True)
    for i in range(num_epochs):
        loss, W, b = update_once(learning_rate, reg_param, W, b)
        if i % 50 == 0:
            print(f"Epoch {i+1}, Loss: {loss}, W norm: {torch.norm(W).item()}")
    return W, b
```

I did not optimize the number of training loops or the learning rate. I experimented with values and used values that showed relatively fast, good convergence. For all models, 751 training loops with a learning rate of 0.1 was more than sufficient for convergence.

[7]:
```
# Try some hyperparameters
update_N(0.1, 0.1, 200, W, b)
print("train_X shape:", train_X_tensor.shape)
print("W shape:", W.shape)
print("b shape:", b.shape)
print("train_Y shape:", train_Y_tensor.shape)
```

```
Epoch 1, Loss: 38893895680.0, W norm: 99413.6015625
Epoch 51, Loss: 1962283776.0, W norm: 94866.2578125
Epoch 101, Loss: 1888352512.0, W norm: 103257.2734375
Epoch 151, Loss: 1869789952.0, W norm: 105281.921875
train_X shape: torch.Size([1022, 40])
W shape: torch.Size([40, 1])
b shape: torch.Size([1])
train_Y shape: torch.Size([1022, 1])
```

[8]:
```
# Create a lot of models with different regularization parameters
reg_params = [0,1e-8,1e-7,1e-6,0.00001,0.0001,0.001,0.01,0.1,1]
model_params = []
for reg in reg_params:
    model_params.append(update_N(0.1, reg, 751, W, b))
```

```
Epoch 1, Loss: 38891413504.0, W norm: 99412.21875
Epoch 51, Loss: 1368936704.0, W norm: 107956.4921875
Epoch 101, Loss: 1140911488.0, W norm: 131148.859375
Epoch 151, Loss: 1064404928.0, W norm: 144762.5
Epoch 201, Loss: 1028435904.0, W norm: 153626.875
Epoch 251, Loss: 1007280896.0, W norm: 159979.25
Epoch 301, Loss: 993184768.0, W norm: 164901.0625
Epoch 351, Loss: 983115520.0, W norm: 168929.328125
Epoch 401, Loss: 975595712.0, W norm: 172346.46875
Epoch 451, Loss: 969792576.0, W norm: 175313.40625
Epoch 501, Loss: 965193344.0, W norm: 177930.421875
Epoch 551, Loss: 961464960.0, W norm: 180265.421875
Epoch 601, Loss: 958382528.0, W norm: 182367.625
Epoch 651, Loss: 955790144.0, W norm: 184274.5
Epoch 701, Loss: 953576768.0, W norm: 186015.5625
Epoch 751, Loss: 951661952.0, W norm: 187614.625
Epoch 1, Loss: 38894858240.0, W norm: 99413.4140625
Epoch 51, Loss: 1368945920.0, W norm: 107955.5859375
Epoch 101, Loss: 1140912256.0, W norm: 131148.484375
Epoch 151, Loss: 1064404288.0, W norm: 144762.375
Epoch 201, Loss: 1028435520.0, W norm: 153626.875
Epoch 251, Loss: 1007280256.0, W norm: 159979.328125
Epoch 301, Loss: 993184128.0, W norm: 164901.1875
Epoch 351, Loss: 983114944.0, W norm: 168929.484375
Epoch 401, Loss: 975595328.0, W norm: 172346.609375
Epoch 451, Loss: 969792256.0, W norm: 175313.578125
Epoch 501, Loss: 965192768.0, W norm: 177930.609375
Epoch 551, Loss: 961464576.0, W norm: 180265.609375
Epoch 601, Loss: 958382080.0, W norm: 182367.796875
Epoch 651, Loss: 955790144.0, W norm: 184274.65625
Epoch 701, Loss: 953576640.0, W norm: 186015.71875
Epoch 751, Loss: 951661632.0, W norm: 187614.796875
Epoch 1, Loss: 38892896256.0, W norm: 99412.9453125
Epoch 51, Loss: 1368941440.0, W norm: 107956.046875
Epoch 101, Loss: 1140912128.0, W norm: 131148.65625
Epoch 151, Loss: 1064404800.0, W norm: 144762.421875
Epoch 201, Loss: 1028436224.0, W norm: 153626.859375
Epoch 251, Loss: 1007281472.0, W norm: 159979.28125
Epoch 301, Loss: 993185408.0, W norm: 164901.078125
Epoch 351, Loss: 983116416.0, W norm: 168929.359375
Epoch 401, Loss: 975596736.0, W norm: 172346.484375
Epoch 451, Loss: 969793664.0, W norm: 175313.40625
Epoch 501, Loss: 965194560.0, W norm: 177930.40625
Epoch 551, Loss: 961466304.0, W norm: 180265.390625
Epoch 601, Loss: 958384000.0, W norm: 182367.59375
Epoch 651, Loss: 955791808.0, W norm: 184274.421875
Epoch 701, Loss: 953578432.0, W norm: 186015.46875
Epoch 751, Loss: 951663488.0, W norm: 187614.53125
Epoch 1, Loss: 38893035520.0, W norm: 99412.1640625
Epoch 51, Loss: 1368945024.0, W norm: 107955.7890625
Epoch 101, Loss: 1140920832.0, W norm: 131148.265625
Epoch 151, Loss: 1064416512.0, W norm: 144761.78125
Epoch 201, Loss: 1028449344.0, W norm: 153625.96875
Epoch 251, Loss: 1007295744.0, W norm: 159978.15625
Epoch 301, Loss: 993200512.0, W norm: 164899.765625
Epoch 351, Loss: 983132032.0, W norm: 168927.84375
Epoch 401, Loss: 975613120.0, W norm: 172344.8125
Epoch 451, Loss: 969810176.0, W norm: 175311.578125
Epoch 501, Loss: 965211392.0, W norm: 177928.421875
Epoch 551, Loss: 961483392.0, W norm: 180263.28125
Epoch 601, Loss: 958401536.0, W norm: 182365.296875
Epoch 651, Loss: 955809344.0, W norm: 184272.0
Epoch 701, Loss: 953596288.0, W norm: 186012.921875
Epoch 751, Loss: 951681728.0, W norm: 187611.84375
Epoch 1, Loss: 38893064192.0, W norm: 99412.4453125
Epoch 51, Loss: 1369011072.0, W norm: 107954.5078125
Epoch 101, Loss: 1141013504.0, W norm: 131145.125
Epoch 151, Loss: 1064525504.0, W norm: 144756.875
Epoch 201, Loss: 1028569600.0, W norm: 153619.40625
Epoch 251, Loss: 1007424832.0, W norm: 159970.0625
Epoch 301, Loss: 993337216.0, W norm: 164890.1875
Epoch 351, Loss: 983274944.0, W norm: 168916.78125
Epoch 401, Loss: 975761344.0, W norm: 172332.28125
Epoch 451, Loss: 969963456.0, W norm: 175297.625
Epoch 501, Loss: 965369216.0, W norm: 177913.046875
Epoch 551, Loss: 961644800.0, W norm: 180246.5
Epoch 601, Loss: 958566208.0, W norm: 182347.171875
Epoch 651, Loss: 955977472.0, W norm: 184252.515625
Epoch 701, Loss: 953767168.0, W norm: 185992.125
Epoch 751, Loss: 951855360.0, W norm: 187589.734375
Epoch 1, Loss: 38893416448.0, W norm: 99412.5703125
Epoch 51, Loss: 1369658368.0, W norm: 107940.5234375
Epoch 101, Loss: 1141948160.0, W norm: 131111.828125
Epoch 151, Loss: 1065629376.0, W norm: 144704.84375
Epoch 201, Loss: 1029788288.0, W norm: 153550.203125
Epoch 251, Loss: 1008732160.0, W norm: 159884.796875
Epoch 301, Loss: 994717184.0, W norm: 164789.5625
Epoch 351, Loss: 984717248.0, W norm: 168801.21875
Epoch 401, Loss: 977257472.0, W norm: 172202.09375
Epoch 451, Loss: 971506688.0, W norm: 175153.046875
Epoch 501, Loss: 966953728.0, W norm: 177754.359375
Epoch 551, Loss: 963267072.0, W norm: 180073.9375
Epoch 601, Loss: 960222080.0, W norm: 182160.984375
Epoch 651, Loss: 957663808.0, W norm: 184053.0
Epoch 701, Loss: 955481472.0, W norm: 185779.453125
Epoch 751, Loss: 953595584.0, W norm: 187364.234375
Epoch 1, Loss: 38892011520.0, W norm: 99411.9140625
Epoch 51, Loss: 1376082432.0, W norm: 107804.578125
Epoch 101, Loss: 1151232512.0, W norm: 130784.25
Epoch 151, Loss: 1076592640.0, W norm: 144191.546875
Epoch 201, Loss: 1041882496.0, W norm: 152866.515625
Epoch 251, Loss: 1021693120.0, W norm: 159042.40625
Epoch 301, Loss: 1008388928.0, W norm: 163795.734375
Epoch 351, Loss: 998990400.0, W norm: 167660.578125
Epoch 401, Loss: 992048448.0, W norm: 170918.0
Epoch 451, Loss: 986748864.0, W norm: 173728.4375
Epoch 501, Loss: 982593664.0, W norm: 176192.078125
Epoch 551, Loss: 979260416.0, W norm: 178376.875
Epoch 601, Loss: 976532928.0, W norm: 180332.046875
Epoch 651, Loss: 974262208.0, W norm: 182095.0
Epoch 701, Loss: 972342592.0, W norm: 183695.203125
Epoch 751, Loss: 970697664.0, W norm: 185156.3125
Epoch 1, Loss: 38894075904.0, W norm: 99412.515625
Epoch 51, Loss: 1439020800.0, W norm: 106465.03125
Epoch 101, Loss: 1240626816.0, W norm: 127603.234375
Epoch 151, Loss: 1180698496.0, W norm: 139270.75
Epoch 201, Loss: 1155384960.0, W norm: 146391.75
Epoch 251, Loss: 1142033920.0, W norm: 151160.359375
Epoch 301, Loss: 1134065920.0, W norm: 154608.671875
Epoch 351, Loss: 1128972288.0, W norm: 157243.578125
Epoch 401, Loss: 1125568896.0, W norm: 159332.46875
Epoch 451, Loss: 1123219072.0, W norm: 161029.4375
Epoch 501, Loss: 1121551872.0, W norm: 162431.3125
Epoch 551, Loss: 1120341248.0, W norm: 163603.75
Epoch 601, Loss: 1119443456.0, W norm: 164593.828125
Epoch 651, Loss: 1118765696.0, W norm: 165436.6875
Epoch 701, Loss: 1118244992.0, W norm: 166159.34375
Epoch 751, Loss: 1117839104.0, W norm: 166782.890625
Epoch 1, Loss: 38892593152.0, W norm: 99412.2734375
Epoch 51, Loss: 1962287104.0, W norm: 94865.5390625
Epoch 101, Loss: 1888351360.0, W norm: 103256.84375
Epoch 151, Loss: 1869788928.0, W norm: 105281.6484375
Epoch 201, Loss: 1860032512.0, W norm: 105430.546875
Epoch 251, Loss: 1853425664.0, W norm: 105035.828125
Epoch 301, Loss: 1848651776.0, W norm: 104519.015625
Epoch 351, Loss: 1845140992.0, W norm: 104015.03125
Epoch 401, Loss: 1842544128.0, W norm: 103563.9765625
Epoch 451, Loss: 1840620160.0, W norm: 103173.546875
Epoch 501, Loss: 1839194112.0, W norm: 102840.3515625
Epoch 551, Loss: 1838136320.0, W norm: 102557.71875
Epoch 601, Loss: 1837352064.0, W norm: 102318.46875
Epoch 651, Loss: 1836770304.0, W norm: 102115.9609375
Epoch 701, Loss: 1836338944.0, W norm: 101944.4296875
Epoch 751, Loss: 1836018944.0, W norm: 101798.9609375
Epoch 1, Loss: 38893019136.0, W norm: 99413.0546875
Epoch 51, Loss: 3726718976.0, W norm: 49885.125
Epoch 101, Loss: 3474076160.0, W norm: 43880.00390625
Epoch 151, Loss: 3413977088.0, W norm: 41812.75
Epoch 201, Loss: 3399680512.0, W norm: 41051.51171875
Epoch 251, Loss: 3396279040.0, W norm: 40743.19921875
Epoch 301, Loss: 3395469568.0, W norm: 40608.23046875
Epoch 351, Loss: 3395277312.0, W norm: 40546.11328125
Epoch 401, Loss: 3395231744.0, W norm: 40516.703125
Epoch 451, Loss: 3395220480.0, W norm: 40502.5703125
Epoch 501, Loss: 3395218432.0, W norm: 40495.7265625
Epoch 551, Loss: 3395217664.0, W norm: 40492.3984375
Epoch 601, Loss: 3395217408.0, W norm: 40490.78515625
Epoch 651, Loss: 3395217664.0, W norm: 40489.99609375
Epoch 701, Loss: 3395217664.0, W norm: 40489.609375
Epoch 751, Loss: 3395217408.0, W norm: 40489.421875
```

[9]:
```python
# Test each model on validation set
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

MSE_scores = []
r2_scores = []
for W, b in model_params:
    predicted_val_Y = val_X.to_numpy() @ W.detach().numpy() + b.detach().numpy()
    MSE_scores.append(mean_squared_error(val_Y, predicted_val_Y))
    r2_scores.append(r2_score(val_Y, predicted_val_Y))
print(val_Y.shape, predicted_val_Y.shape)
```

```
(438,) (438, 1)
```

With a regularization hyperparameter of 0 (meaning no L2 regularization), the MSE score was 9.593*10^8 and the R^2 score was 0.8625.

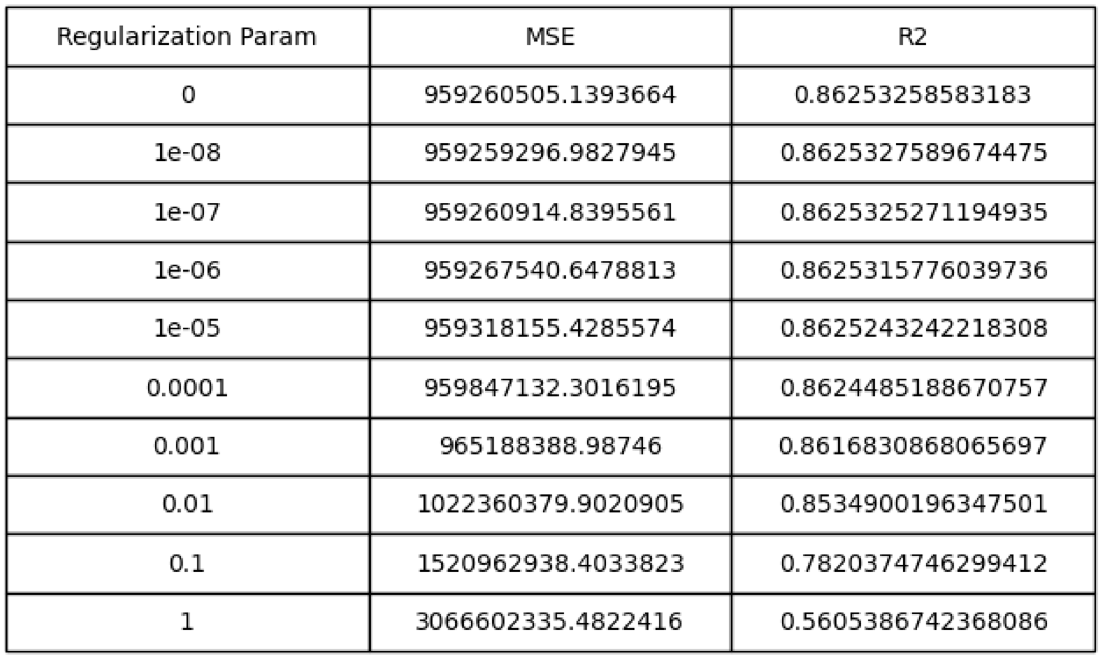
1.3 Regularization hyperparameter optimization

For the remaining hyperparameter of the regularization constant, I evaluated the resulting models of various regularization constants. I did this by using the gradient descent method on the training set and evaluating on the validation set. Evaluation was based on maximizing the R^2 score, but the MSE score was also observed to decrease with this optimization.

I did this optimization once to determine the best regularization hyperparameter to be between 1e-8 and 1e-6.

[10]:
```python
# Visualize metrics
import matplotlib.pyplot as plt

# This is adapted code from HW1
# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how do i create a table"
models_names = reg_params
mses = MSE_scores
r2s = r2_scores

table_data = list(zip(models_names, mses, r2s))
fig, ax = plt.subplots(figsize=(8,3))
ax.axis('off')  # hide axes

# Create the table
table = ax.table(cellText=table_data,
                 colLabels=['Regularization Param', 'MSE', 'R2'],
                 cellLoc='center',
                 loc='center')

table.auto_set_font_size(True)
table.scale(1, 2)
```

```
plt.show()
```



Regularization Param	MSE	R2
0	959260505.1393664	0.86253258583183
1e-08	959259296.9827945	0.8625327589674475
1e-07	959260914.8395561	0.8625325271194935
1e-06	959267540.6478813	0.8625315776039736
1e-05	959318155.4285574	0.8625243242218308
0.0001	959847132.3016195	0.8624485188670757
0.001	965188388.98746	0.8616830868065697
0.01	1022360379.9020905	0.8534900196347501
0.1	1520962938.4033823	0.7820374746299412
1	3066602335.4822416	0.5605386742368086

Then I did the optimization again for regularization constants between 1e-8 and 1e-6 (Figure 3). The final best model was determined to use a regularization parameter of 1e-8. For the final model, the MSE score was (9.593*10^8) and the (R^2) score was 0.8625.

[11]:
```
# Try some more regularization parameters
reg_params = [1e-8, 2.5e-8,5e-8,7.5e-8,1e-7,2.5e-7,5e-7,7.5e-7,1e-6]
model_params = []
for reg in reg_params:
    model_params.append(update_N(0.1, reg, 751, W, b))
```

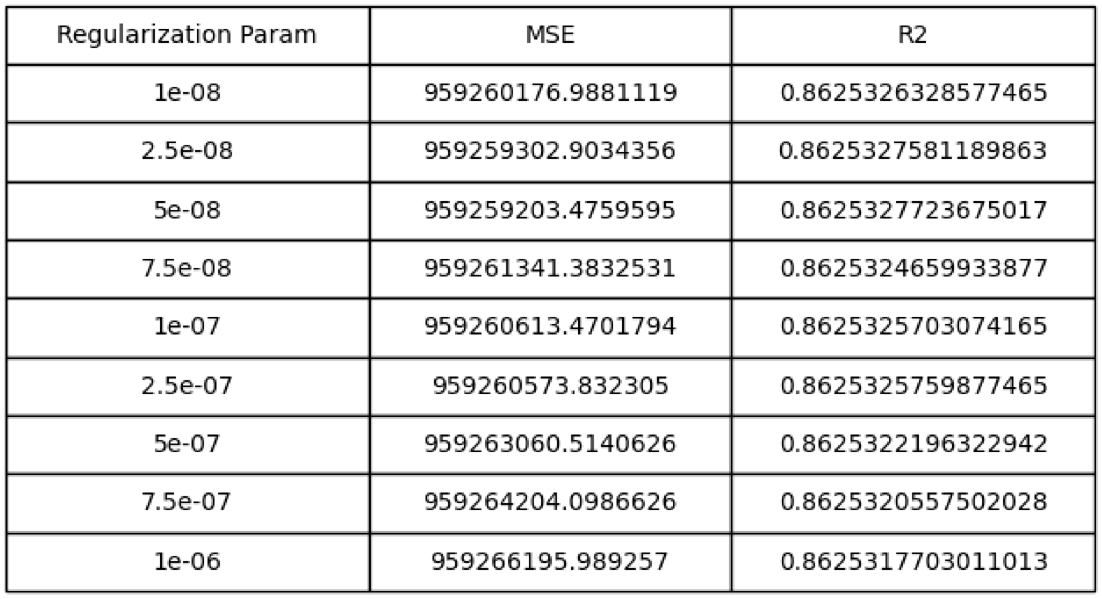
```
Epoch 1, Loss: 38892236800.0, W norm: 99412.1015625
Epoch 51, Loss: 1368933632.0, W norm: 107956.2890625
Epoch 101, Loss: 1140909184.0, W norm: 131148.8125
Epoch 151, Loss: 1064403392.0, W norm: 144762.5
Epoch 201, Loss: 1028434752.0, W norm: 153626.890625
Epoch 251, Loss: 1007280064.0, W norm: 159979.28125
Epoch 301, Loss: 993184128.0, W norm: 164901.109375
Epoch 351, Loss: 983115072.0, W norm: 168929.390625
Epoch 401, Loss: 975595584.0, W norm: 172346.515625
Epoch 451, Loss: 969792576.0, W norm: 175313.4375
Epoch 501, Loss: 965193152.0, W norm: 177930.421875
Epoch 551, Loss: 961465024.0, W norm: 180265.421875
Epoch 601, Loss: 958382720.0, W norm: 182367.609375
Epoch 651, Loss: 955790400.0, W norm: 184274.46875
Epoch 701, Loss: 953577024.0, W norm: 186015.515625
Epoch 751, Loss: 951662080.0, W norm: 187614.578125
Epoch 1, Loss: 38894239744.0, W norm: 99413.1171875
Epoch 51, Loss: 1368943104.0, W norm: 107955.6796875
Epoch 101, Loss: 1140911232.0, W norm: 131148.5
Epoch 151, Loss: 1064403776.0, W norm: 144762.375
Epoch 201, Loss: 1028435008.0, W norm: 153626.84375
Epoch 251, Loss: 1007280128.0, W norm: 159979.28125
Epoch 301, Loss: 993184064.0, W norm: 164901.140625
Epoch 351, Loss: 983115008.0, W norm: 168929.4375
Epoch 401, Loss: 975595648.0, W norm: 172346.5625
Epoch 451, Loss: 969792448.0, W norm: 175313.5
Epoch 501, Loss: 965193024.0, W norm: 177930.515625
Epoch 551, Loss: 961464896.0, W norm: 180265.515625
Epoch 601, Loss: 958382656.0, W norm: 182367.71875
Epoch 651, Loss: 955790208.0, W norm: 184274.5625
Epoch 701, Loss: 953577024.0, W norm: 186015.625
Epoch 751, Loss: 951662144.0, W norm: 187614.671875
Epoch 1, Loss: 38891732992.0, W norm: 99411.8515625
Epoch 51, Loss: 1368942592.0, W norm: 107955.8046875
Epoch 101, Loss: 1140912768.0, W norm: 131148.453125
Epoch 151, Loss: 1064405056.0, W norm: 144762.234375
Epoch 201, Loss: 1028436032.0, W norm: 153626.71875
Epoch 251, Loss: 1007280896.0, W norm: 159979.171875
Epoch 301, Loss: 993184832.0, W norm: 164901.03125
Epoch 351, Loss: 983115456.0, W norm: 168929.34375
Epoch 401, Loss: 975595904.0, W norm: 172346.5
Epoch 451, Loss: 969792896.0, W norm: 175313.4375
Epoch 501, Loss: 965193600.0, W norm: 177930.453125
Epoch 551, Loss: 961465280.0, W norm: 180265.453125
Epoch 601, Loss: 958383104.0, W norm: 182367.65625
Epoch 651, Loss: 955790720.0, W norm: 184274.515625
Epoch 701, Loss: 953577408.0, W norm: 186015.5625
Epoch 751, Loss: 951662656.0, W norm: 187614.625
Epoch 1, Loss: 38893068288.0, W norm: 99412.296875
Epoch 51, Loss: 1368943744.0, W norm: 107955.5625
Epoch 101, Loss: 1140913280.0, W norm: 131148.28125
Epoch 151, Loss: 1064406144.0, W norm: 144762.109375
Epoch 201, Loss: 1028437184.0, W norm: 153626.53125
Epoch 251, Loss: 1007282368.0, W norm: 159978.9375
Epoch 301, Loss: 993186432.0, W norm: 164900.734375
Epoch 351, Loss: 983117312.0, W norm: 168929.0
Epoch 401, Loss: 975597696.0, W norm: 172346.09375
Epoch 451, Loss: 969794560.0, W norm: 175313.046875
Epoch 501, Loss: 965195136.0, W norm: 177930.03125
Epoch 551, Loss: 961466816.0, W norm: 180265.015625
Epoch 601, Loss: 958384448.0, W norm: 182367.203125
Epoch 651, Loss: 955792192.0, W norm: 184274.0625
Epoch 701, Loss: 953578880.0, W norm: 186015.109375
Epoch 751, Loss: 951663808.0, W norm: 187614.15625
Epoch 1, Loss: 38892433408.0, W norm: 99411.8984375
Epoch 51, Loss: 1368939264.0, W norm: 107955.90625
Epoch 101, Loss: 1140911744.0, W norm: 131148.53125
Epoch 151, Loss: 1064405184.0, W norm: 144762.265625
Epoch 201, Loss: 1028436544.0, W norm: 153626.703125
Epoch 251, Loss: 1007281728.0, W norm: 159979.125
Epoch 301, Loss: 993185600.0, W norm: 164900.9375
Epoch 351, Loss: 983116544.0, W norm: 168929.21875
Epoch 401, Loss: 975597184.0, W norm: 172346.34375
Epoch 451, Loss: 969793920.0, W norm: 175313.296875
Epoch 501, Loss: 965194688.0, W norm: 177930.296875
Epoch 551, Loss: 961466368.0, W norm: 180265.265625
Epoch 601, Loss: 958384128.0, W norm: 182367.453125
Epoch 651, Loss: 955791936.0, W norm: 184274.3125
Epoch 701, Loss: 953578496.0, W norm: 186015.375
Epoch 751, Loss: 951663552.0, W norm: 187614.421875
Epoch 1, Loss: 38893830144.0, W norm: 99413.2890625
Epoch 51, Loss: 1368944000.0, W norm: 107956.140625
Epoch 101, Loss: 1140915456.0, W norm: 131148.625
Epoch 151, Loss: 1064407936.0, W norm: 144762.3125
Epoch 201, Loss: 1028439040.0, W norm: 153626.703125
Epoch 251, Loss: 1007284032.0, W norm: 159979.09375
Epoch 301, Loss: 993188032.0, W norm: 164900.890625
Epoch 351, Loss: 983118848.0, W norm: 168929.140625
Epoch 401, Loss: 975599232.0, W norm: 172346.265625
Epoch 451, Loss: 969796224.0, W norm: 175313.15625
Epoch 501, Loss: 965197120.0, W norm: 177930.15625
Epoch 551, Loss: 961468800.0, W norm: 180265.109375
Epoch 601, Loss: 958386432.0, W norm: 182367.296875
Epoch 651, Loss: 955794432.0, W norm: 184274.140625
Epoch 701, Loss: 953581120.0, W norm: 186015.171875
Epoch 751, Loss: 951666432.0, W norm: 187614.203125
Epoch 1, Loss: 38891380736.0, W norm: 99411.3828125
Epoch 51, Loss: 1368943872.0, W norm: 107955.8046875
Epoch 101, Loss: 1140918656.0, W norm: 131148.171875
Epoch 151, Loss: 1064412800.0, W norm: 144761.71875
Epoch 201, Loss: 1028444352.0, W norm: 153625.984375
Epoch 251, Loss: 1007289344.0, W norm: 159978.296875
Epoch 301, Loss: 993193408.0, W norm: 164900.03125
Epoch 351, Loss: 983124160.0, W norm: 168928.234375
Epoch 401, Loss: 975604608.0, W norm: 172345.328125
Epoch 451, Loss: 969801664.0, W norm: 175312.234375
Epoch 501, Loss: 965202432.0, W norm: 177929.171875
Epoch 551, Loss: 961474240.0, W norm: 180264.15625
Epoch 601, Loss: 958392064.0, W norm: 182366.28125
Epoch 651, Loss: 955799616.0, W norm: 184273.109375
Epoch 701, Loss: 953586368.0, W norm: 186014.140625
Epoch 751, Loss: 951671616.0, W norm: 187613.15625
Epoch 1, Loss: 38892634112.0, W norm: 99412.09375
Epoch 51, Loss: 1368944640.0, W norm: 107955.828125
Epoch 101, Loss: 1140917888.0, W norm: 131148.421875
Epoch 151, Loss: 1064412928.0, W norm: 144762.015625
Epoch 201, Loss: 1028445440.0, W norm: 153626.328125
Epoch 251, Loss: 1007291456.0, W norm: 159978.578125
Epoch 301, Loss: 993196096.0, W norm: 164900.25
Epoch 351, Loss: 983127424.0, W norm: 168928.375
Epoch 401, Loss: 975608192.0, W norm: 172345.390625
Epoch 451, Loss: 969805504.0, W norm: 175312.1875
Epoch 501, Loss: 965206720.0, W norm: 177929.046875
Epoch 551, Loss: 961478656.0, W norm: 180263.921875
Epoch 601, Loss: 958396480.0, W norm: 182365.984375
Epoch 651, Loss: 955804672.0, W norm: 184272.703125
Epoch 701, Loss: 953591488.0, W norm: 186013.65625
Epoch 751, Loss: 951676736.0, W norm: 187612.609375
Epoch 1, Loss: 38893563904.0, W norm: 99412.609375
Epoch 51, Loss: 1368955520.0, W norm: 107955.2578125
Epoch 101, Loss: 1140922752.0, W norm: 131148.0
Epoch 151, Loss: 1064416256.0, W norm: 144761.75
Epoch 201, Loss: 1028448704.0, W norm: 153626.046875
Epoch 251, Loss: 1007294976.0, W norm: 159978.3125
Epoch 301, Loss: 993199872.0, W norm: 164899.984375
Epoch 351, Loss: 983131328.0, W norm: 168928.0625
Epoch 401, Loss: 975612352.0, W norm: 172345.03125
Epoch 451, Loss: 969809728.0, W norm: 175311.796875
Epoch 501, Loss: 965211008.0, W norm: 177928.640625
Epoch 551, Loss: 961483008.0, W norm: 180263.46875
Epoch 601, Loss: 958401024.0, W norm: 182365.5
Epoch 651, Loss: 955809088.0, W norm: 184272.203125
Epoch 701, Loss: 953595968.0, W norm: 186013.09375
Epoch 751, Loss: 951681472.0, W norm: 187612.015625
```

[12]:
```python
# Define reusable compute and graph function
def compute_and_graph_scores():
    MSE_scores = []
    r2_scores = []
    for W, b in model_params:
        predicted_val_Y = val_X.to_numpy() @ W.detach().numpy() + b.detach().
 ↪numpy()
        MSE_scores.append(mean_squared_error(val_Y, predicted_val_Y))
        r2_scores.append(r2_score(val_Y, predicted_val_Y))
    print(val_Y.shape, predicted_val_Y.shape)
    # This is adapted code from HW1
    # Adapted from code generated by ChatGPT (OpenAI, 2025)
    # Prompt: "how do i create a table"
    models_names = reg_params
    mses = MSE_scores
    r2s = r2_scores

    table_data = list(zip(models_names, mses, r2s))
    fig, ax = plt.subplots(figsize=(8,3))
    ax.axis('off')  # hide axes

    # Create the table
    table = ax.table(cellText=table_data,
                     colLabels=['Regularization Param', 'MSE', 'R2'],
                     cellLoc='center',
                     loc='center')

    table.auto_set_font_size(True)
    table.scale(1, 2)
    plt.show()
```

[13]:

```python
# Compute and graph scores
compute_and_graph_scores()
```

```
(438,) (438, 1)
```



Regularization Param	MSE	R2
1e-08	959260176.9881119	0.8625326328577465
2.5e-08	959259302.9034356	0.8625327581189863
5e-08	959259203.4759595	0.8625327723675017
7.5e-08	959261341.3832531	0.8625324659933877
1e-07	959260613.4701794	0.8625325703074165
2.5e-07	959260573.832305	0.8625325759877465
5e-07	959263060.5140626	0.8625322196322942
7.5e-07	959264204.0986626	0.8625320557502028
1e-06	959266195.989257	0.8625317703011013

1.4 Re-evaluation of OLS

I re-evaluated the scikit-learn analytic solution by training the model on the training set and evaluating scores on the validation set. This was necessary, because HW1 did not split the training data into training and validation sets.

The MSE score was 9.223*10^8 and the R^2 score was 0.8678.

```
[14]: # Compare to sklearn
      from sklearn.linear_model import LinearRegression

      sk_model = LinearRegression()
      sk_model.fit(train_X, train_Y)
      print("Sklearn R2:", sk_model.score(val_X, val_Y))
```

```
Sklearn R2: 0.8678334659150575
```

1.5 Discussion of optimization methods

There are three optimization methods to compare: OLS closed-form (OLS), gradient descent (GD), and gradient descent with L2 regularization (GDL2). Here is a summary of the evaluation metrics for the best model produced for these methods.

```
[15]: # Compare best SKLearn and autograd
      print("Sklearn R2:", sk_model.score(val_X, val_Y))
      print("Autograd R2:", max(r2_scores))
      print("Sklearn MSE:", mean_squared_error(val_Y, sk_model.predict(val_X)))
      print("Autograd MSE:", min(MSE_scores))
```

```
Sklearn R2: 0.8678334659150575
Autograd R2: 0.8625327589674475
Sklearn MSE: 922270466.9031091
Autograd MSE: 959259296.9827945
```

OLS) MSE score = (9.223*10^8) and (R^2) score = 0.8678.

GD) MSE score = (9.59261*10^8) and (R^2) score = 0.8625325.

GDL2) MSE score = (9.59260*10^8) and (R^2) score = 0.8625327.

First, comparing OLS to both GD and GDL2, the analytic, closed form OLS solution proves to give the best performing model in terms of both (R^2) and MSE. However, the difference is not too much.

Comparing GD to GDL2, we see an extremely marginal improvement in applying L2 regularization to gradient descent. This informs that the vanilla GD method does not easily overfit for this dataset.

While OLS proves to be the dominant method for this dataset, not all linear regressions should rely on the closed-form solution. One instance is when the closed-form solution cannot be calculated because the design matrix, X (the matrix of features and samples), is not invertible. X would be non-invertible when X is not full rank, like when there are highly collinear features. Here, we show that GD and GDL2 are decent alternatives to OLS, so one can use these gradient descent methods when X is non-invertible.

On a conceptual level, the analytic solution fails when X in non-invertible because this implies that there are multiple (infinite) critical points on the loss function. On the other hand, gradient descent methods can work around local critical points generated from co-linearity by using a weighted average gradient descent.

distribution-convergence

October 6, 2025

```
[1]: # Data generating distribution and convergence of linear regression
```

```
[1]: # Run this code to see markdowns easier
     # Adapted from code generated by ChatGPT (OpenAI, 2025)
     # Prompt: "how to make markdown cells a different color in jupyter notebooks"
     from IPython.display import Javascript

     Javascript("""
     var style = document.createElement('style');
     style.innerHTML = `
       /* Target Markdown cells only */
       .jp-Notebook .jp-MarkdownOutput {
           background-color: #d8f0ff !important;
           border: none !important;           /* remove border */
           border-radius: 10px;
           padding: 12px !important;
           box-shadow: 0 0 8px rgba(112, 174, 224, 0.3);
       }
     `;
     document.head.appendChild(style);
     """)
```

```
[1]: <IPython.core.display.Javascript object>
```

1 Data Generating Distribution and Convergence of Linear Regression

1.1 Generating synthetic dataset

```
[9]: # Experiment with numpy.random.normal
     import numpy as np
     a = np.random.normal(168, 30, size=2)
     print(a)
     a.shape
```

```
[149.12357326 110.03730174]
```

```
[9]: (2,)
```

I generated a synthetic dataset based on a linear model with additive Gaussian noise using the following data generating process:

```
Y = alpha + beta X + epsilon
```

Where each realization of   is a single random value drawn from a normal distribution with mean 0 and standard deviation 20.

To generate a random input space, I generated these X values from a normal distribution as well. Each realization of X is also a single random value drawn from a normal distribution with a mean 168 and standard deviation 30.

Most importantly, I set alpha = 20, and beta = 0.5.

[17]:
```
# Generate one synthetic dataset
number_of_samples = 10
alpha = 20
beta = 0.5
X = np.random.normal(168, 30, size=number_of_samples).reshape(-1,1)
Y = alpha + beta * X + np.random.normal(0, 20, size=number_of_samples).
 ↪reshape(-1,1)
Y.shape
Y
```

[17]:
```
array([[114.86515818],
       [154.85444668],
       [173.98868108],
       [110.79627741],
       [121.36077989],
       [ 91.25897211],
       [ 93.46414028],
       [105.02717971],
       [ 89.33269864],
       [106.35165489]])
```

1.2 Running linear regression on various sample sizes

I generated datasets using the data generating process of sample size n = 10^2, 10^3, 10^4, 10^5, 10^6. Then I ran a linear regression on each of these datasets.

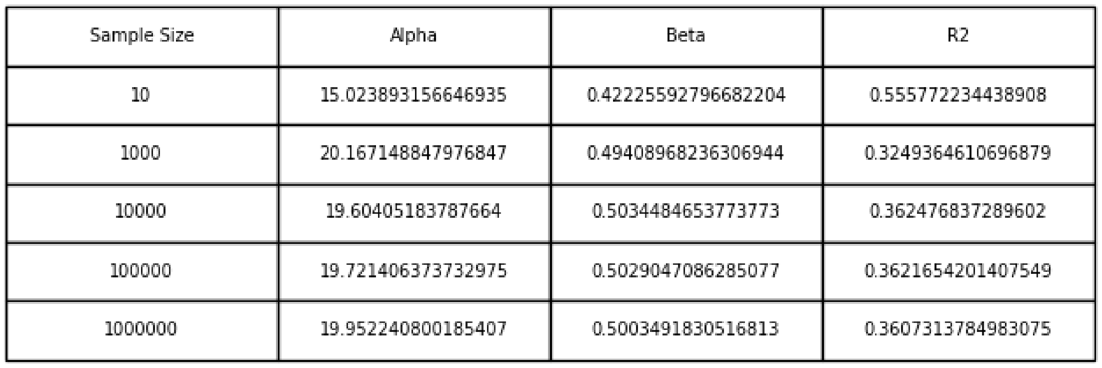
[19]:
```
# Function to generate any n-size dataset
def generate_dataset(number_of_samples=10):
    alpha = 20
    beta = 0.5
    X = np.random.normal(168, 30, size=number_of_samples).reshape(-1,1)
    Y = alpha + beta * X + np.random.normal(0, 20, size=number_of_samples).
 ↪reshape(-1,1)
    return X,Y
```

[23]:
```
# Generate datasets
datasets = {}
sample_sizes = [10,10**3,10**4,10**5,10**6]
for size in sample_sizes:
    datasets[size] = generate_dataset(size)
datasets[1000][0].shape
```

[23]:
```
(1000, 1)
```

[30]:
```
# Define linear regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

def model(X,Y):
    sk_model = LinearRegression()
    sk_model.fit(X, Y)
    r2 = sk_model.score(X, Y)
    return sk_model.coef_.item(), sk_model.intercept_.item(), r2
```

[32]:
```
# Run linear regressions
# This is adapted code from HW1
# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how do i create a table"
import matplotlib.pyplot as plt

sizes = sample_sizes
alphas = []
betas = []
r2s = []
for size, (X, Y) in datasets.items():
    b_calc, a_calc, r2 = model(X,Y)
    alphas.append(a_calc)
    betas.append(b_calc)
    r2s.append(r2)

table_data = list(zip(sizes, alphas, betas, r2s))
fig, ax = plt.subplots(figsize=(8,3))
ax.axis('off')  # hide axes

# Create the table
table = ax.table(cellText=table_data,
                 colLabels=['Sample Size', 'Alpha', 'Beta', 'R2'],
                 cellLoc='center',
                 loc='center')

table.auto_set_font_size(True)
table.scale(1, 2)
plt.show()
```



Sample Size	Alpha	Beta	R2
10	15.023893156646935	0.42225592796682204	0.555772234438908
1000	20.167148847976847	0.49408968236306944	0.3249364610696879
10000	19.60405183787664	0.5034484653773773	0.362476837289602
100000	19.721406373732975	0.5029047086285077	0.3621654201407549
1000000	19.952240800185407	0.5003491830516813	0.3607313784983075

1.3 Discussion of convergence

Running linear regression on increasing sample sizes shows that the weight coefficient converges close to 20 and the bias coefficient converges to 0.5. This shows that the calculated model coefficients converge to the actual coefficients of alpha and beta used in the true data generating process. It seems like the R^2 score converges to 0.36, not 0.

```
[ ]: # Observations
     # The Alpha converges to 20 and the Beta converges to 0.5
     # These are the alpha and beta values we chose during the data generating␣
      ↪process

     # The R2 converges to 0.36, not 0.
```

1.4 Convergence of R^2

With the assumption that the model parameters for alpha and beta converge to the true data generating values, we can show that the R^2 score converges to (1 − var(epsilon)/var(Y).

② ⓓ

$$\boxed{\begin{aligned} var(\mathcal{E}) &= \frac{1}{n}\sum_{i=1}^{n}\left(\mathcal{E}^{(i)} - \bar{\mathcal{E}}\right)^2 \\ var(Y) &= \frac{1}{n}\sum_{i=1}^{n}\left(Y^{(i)} - \bar{Y}\right)^2 \end{aligned}}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{n}\left(y^{(i)} - \bar{y}\right)^2}$$

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \bar{y}\right)^2}$$

$Y \sim \alpha + \beta X + \mathcal{E}$

$\hat{Y} = \alpha_{calc} + \beta_{calc} X$

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{var(Y)}$$

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(\alpha + \beta x^{(i)} + \mathcal{E}^{(i)} - [\alpha_{calc} + \beta_{calc} X^{(i)}]\right)^2}{var(Y)}$$

assume $\alpha_{calc} = \alpha$ $\beta_{calc} = \beta$

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(\cancel{\alpha} - \cancel{\alpha} + \cancel{\beta x^{(i)}} - \cancel{\beta x^{(i)}} + \mathcal{E}^{(i)}\right)^2}{var(Y)}$$

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\mathcal{E}^{(i)2}}{var(Y)}$$

$\bar{\mathcal{E}} = 0$ so...

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}\left(\mathcal{E}^{(i)} - \bar{\mathcal{E}}\right)^2}{var(Y)} = \boxed{1 - \frac{var(\mathcal{E})}{var(Y)}}$$

1.5 Alternative models for R2 convergence

In the above section, it was shown that an appropriate linear model will converge to (1 − var(epsilon)/var(Y)). These values are only dependent on the properties of the data generating distribution itself, and no score loss is contributed by model parameters. So it is mathematically impossible to improve from this inherent noise of the data points themselves. Thus, we cannot improve the asymptotic R^2 score.

In other words, the optimized linear model got as close to the data generating distribution as possible, with the exception of random additive gaussian noise. We saw this in the R^2 score derivation, where the numerator canceled out all terms except epsilon. Trying to deviate and get a "better" model will only add more separation between the predictions and the non-additive portion of the data generating distribution.

1.6 Important characteristics of data generating distribution

As seen in this example, the R^2 score depends on $1 - \text{var(epsilon)}/\text{var}(Y))$. This means that a "poor" model will emerge if the variance of the error is extremely high compared to the variance of the label; this occurs when the noise variance is high compared to the variance in the input.

Thus, a "good" model with good evaluation metrics also depends on the underlying data generating distribution having a combination of narrow noise variance and wide distribution of inputs.

Additionally, beyond a good evaluation metric, we would ideally have a large sample size so that the model parameters can converge towards the true data generating constants.

[ ]:
```

```

NLP-disaster-tweets

October 6, 2025

```
[1]: # Binary Classification on Text Data
```

```
[5]: # Run this code to see markdowns easier
# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how to make markdown cells a different color in jupyter notebooks"
from IPython.display import Javascript

Javascript("""
var style = document.createElement('style');
style.innerHTML = `
  /* Target Markdown cells only */
  .jp-Notebook .jp-MarkdownOutput {
      background-color: #d8f0ff !important;
      border: none !important;          /* remove border */
      border-radius: 10px;
      padding: 12px !important;
      box-shadow: 0 0 8px rgba(112, 174, 224, 0.3);
  }
`;
document.head.appendChild(style);
""")
```

```
[5]: <IPython.core.display.Javascript object>
```

0.0.1 Test markdown

This should have a blue background now.

1 Binary Classification on Text Data

1.1 Download the data

```
[26]: # Imports
import pandas as pd

train_presplit = pd.read_csv("./data-from-NLP-disaster-tweets/train.csv")
test = pd.read_csv("./data-from-NLP-disaster-tweets/test.csv")
print(train_presplit.shape)
```

```
test[10:20]
```

```
(7613, 5)
```

[26]:
```
    id keyword                        location  \
10  30     NaN                             NaN
11  35     NaN                             NaN
12  42     NaN                             NaN
13  43     NaN                             NaN
14  45     NaN                             NaN
15  46  ablaze                          London
16  47  ablaze  Niall's place | SAF 12 SQUAD |
17  51  ablaze                         NIGERIA
18  58  ablaze                 Live On Webcam
19  60  ablaze         Los Angeles, Califnordia

                                                 text
10                                No I don't like cold!
11                           NOOOOOOOOO! Don't do that!
12                               No don't tell me that!
13                                            What if?!
14                                             Awesome!
15  Birmingham Wholesale Market is ablaze BBC News…
16  @sunkxssedharry will you wear shorts for race …
17  #PreviouslyOnDoyinTv: Toke Makinwa Ûªs marriag…
18  Check these out: http://t.co/rOI2NSmEJJ http:/…
19  PSA: I Ûªm splitting my personalities.\n\n?? t…
```

The Kaggle NLP with Disaster Tweets dataset was downloaded. There are 7613 training data samples and 3263 testing data points. Of the 7613 training data points, 3271, or 43%, are tweets about disasters. The remaining 4342, or 57%, are tweets about nondisasters (binary classification).

[9]:
```python
# Data exploration
print(train_presplit.shape)
print(test.shape)
```

```
(7613, 5)
(3263, 4)
```

[12]:
```python
# Percent of training tweets are disasters
count_disasters = (train_presplit["target"] == 1).sum()
print(count_disasters)
```

```
3271
```

1.2 Split the training data

The training data was split into a training set and validation set using a random 70-30 split.

[8]:
```
# Split training data
from sklearn.model_selection import train_test_split

train, validate = train_test_split(
    train_presplit, test_size=0.3, random_state=42, shuffle=True
)
print(train.shape)
train.head()
```

```
(5329, 5)
```

[8]:
```
         id            keyword                 location  \
1186   1707  bridge%20collapse                      NaN
4071   5789               hail  Carol Stream, Illinois
5461   7789             police                 Houston
5787   8257            rioting                      NaN
7445  10656             wounds          Lake Highlands

                                                   text  target
1186  Ashes 2015: Australia Ûªs collapse at Trent Br…       0
4071  GREAT MICHIGAN TECHNIQUE CAMP\nB1G THANKS TO @…       1
5461  CNN: Tennessee movie theater shooting suspect …       1
5787  Still rioting in a couple of hours left until …       1
7445  Crack in the path where I wiped out this morni…       0
```

1.3 Preprocessing

The following preprocessing steps were taken on the tweet text data.

1) Convert all words to lowercase.
2) Lemmatize all words.
3) Strip punctuations (but keep @s, #s)
4) Handle @username and #topic tags
5) Strip stop words
6) Strip Û symbols and other non-standard characters
7) Strip link details (but keep http and https)

The following section will describe each preprocessing step.

1.3.1 Lowercase

All words were converted to lowercase. This was automatically decided because "hello" and "Hello" have the same semantic meaning. While there may be some semantic differences due to capitalization, like "Apple" and "apple", most capitalization differences are only driven by sentence start capitalizations.

1.3.2 Lemmatize

Lemmatizing means to turn all words into their root. This was done to converge words into their semantic meaning. This was done with nltk.stem. It is important to lemmatize words to properly group semantically same words. If we kept words nonlemmatized, then our feature set would be incredibly high cardinality (leading to memory issues) and less represented per feature (potential overfit issue).

1.3.3 Punctuations

Punctuations were also removed since “apple.”, “apple,” and “apple” all have the same semantic meaning. So we can lower cardinality and improve representation per feature like the lemmatizing step.

1.3.4 Twitter tags

Particularly with Twitter tweets, many text had @username or #topic attached. I compared the frequency of @ and # tags on disaster vs nondisaster tweets. For @username tags, found that 20.7% of disaster tweets had # tags and 31.4% of nondisaster tweets had # tags. For #topic tags, found that 26.8% of disaster tweets had # tags and 20.4% of nondisaster tweets had # tags. I thought these differences were significant enough that I kept the tags as tokens. So the data set would identify if a text contained @ or #.

For @username tags, the username portion is not semantically relevant, so I removed the username. However, for #topic, the topic text is very relevant to the tweet, so I kept it (but separated it from # as its own token).

1.3.5 Strip stop words

All common stop words were removed. Stop words are semantically empty and common words like “the”, “a”, and “me”. Once again, removing semantically irrelevant words reduces feature cardinality and improves the number of samples for each feature. These positively improve issues in memory and overfitting.

[22]:
```python
# Preprocess decision-making
# YES: Convert all words to lowercase
# YES: Lemmatize all words
# YES: Strip punctuation
# YES: Strip stop words

# Maybe: Strip @s
count_ats = train_presplit[train_presplit["text"].str.contains("@")]
count_ats_disaster = (count_ats["target"] == 1).sum() /␣
 ↪(train_presplit["target"] == 1).sum()
count_ats_nondisaster = (count_ats["target"] == 0).sum() /␣
 ↪(train_presplit["target"] == 0).sum()
print(f"% Disasters with @: {count_ats_disaster}, % Nondisasters with @:␣
 ↪{count_ats_nondisaster}")
```

```
% Disasters with @: 0.20666462855395903, % Nondisasters with @:
0.31391064025794563
```

[21]:
```
# Maybe: Strip #s
count_ats = train_presplit[train_presplit["text"].str.contains("#")]
count_ats_disaster = (count_ats["target"] == 1).sum() /␣
 ↪(train_presplit["target"] == 1).sum()
count_ats_nondisaster = (count_ats["target"] == 0).sum() /␣
 ↪(train_presplit["target"] == 0).sum()
print(f"% Disasters with #: {count_ats_disaster}, % Nondisasters with #:␣
 ↪{count_ats_nondisaster}")
```

```
% Disasters with #: 0.2675022928767961, % Nondisasters with #:
0.20405343159834177
```

1.3.6 Strip non-standard characters

There are some odd square-Û symbols in the dataset that likely correspond to emojis. However, it seems that all non-standard characters were recorded the same way in the Kaggle dataset, so all the different emoji semantics got lost. We only keep the fact that an emoji or some other non-standard character was used. Since the semantic meaning of emojis were lost, I removed these non-standard characters.

[27]:
```
# Maybe: Strip  Û symbols that did not get recorded properly
count_ats = train_presplit[train_presplit["text"].str.contains(" Û")]
count_ats_disaster = (count_ats["target"] == 1).sum() /␣
 ↪(train_presplit["target"] == 1).sum()
count_ats_nondisaster = (count_ats["target"] == 0).sum() /␣
 ↪(train_presplit["target"] == 0).sum()
print(f"% Disasters with  Û: {count_ats_disaster}, % Nondisasters with  Û:␣
 ↪{count_ats_nondisaster}")
```

```
% Disasters with  Û: 0.09110363803118313, % Nondisasters with  Û:
0.07231690465223399
```

1.3.7 Strip link details

The Kaggle dataset also included links. The link details are not useful to the model, because the url is not lemmatizable to a semantic root nor is it common for the same url to be repeated across many tweets. However, I was curious if the presence of a link itself was different between tweet classes. For http: links, 62.9% of disaster tweets included a link while only 35.7% of nondisaster tweets included a link. For https: links, 4.1% of disaster tweets included a link while 6.3% of nondisaster tweets included a link. I thought these differences were significant enough to keep the http or https as a token (thus allowing the model to know if a link was included). I decided to tokenize the two link types separately due to the different distributions seen (http: links are more popular for disaster tweets and https: links are more popular for nondisasater tweets); this also makes sense, since https: links have different rules than http: links.

[34]:
```
# Maybe: Strip urls
count_ats = train_presplit[train_presplit["text"].str.contains("http:")]
count_ats_disaster = (count_ats["target"] == 1).sum() /␣
 ↪(train_presplit["target"] == 1).sum()
count_ats_nondisaster = (count_ats["target"] == 0).sum() /␣
 ↪(train_presplit["target"] == 0).sum()
print(f"% Disasters with http: {count_ats_disaster}, % Nondisasters with http:␣
 ↪{count_ats_nondisaster}")
```

```
% Disasters with http: 0.6285539590339346, % Nondisasters with http:
0.35651773376324275
```

[32]:
```
# Maybe: Strip urls HTTPS
count_ats = train_presplit[train_presplit["text"].str.contains("https")]
count_ats_disaster = (count_ats["target"] == 1).sum() /␣
 ↪(train_presplit["target"] == 1).sum()
count_ats_nondisaster = (count_ats["target"] == 0).sum() /␣
 ↪(train_presplit["target"] == 0).sum()
print(f"% Disasters with https: {count_ats_disaster}, % Nondisasters with https:
 ↪ {count_ats_nondisaster}")
```

```
% Disasters with https: 0.040966065423417915, % Nondisasters with https:
0.06287425149700598
```

[ ]:
```
# Final preprocess list
# Convert all words to lowercase
# Lemmatize all words
# Strip punctuation (but keep @s, #s)
# Strip stop words
# Strip Û symbols and other non-standard characters
# Strip link specifics BUT keep the fact that a link was included (And also␣
 ↪distinguishing between http: and https:)
```

[94]:
```
# Preprocessing function definitions
from nltk.stem import *
import nltk
from nltk.tokenize import word_tokenize
import re

# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how to use nltk stem to take a string with multiple words and␣
 ↪convert them to the stems"
# Prompt: "how to only keep abc characters in a string"
# Prompt: "create a set of common stop words like the, and, or"

nltk.download("punkt")
nltk.download('punkt_tab')
ps = PorterStemmer()
```

```
common_stops = {
    # Standard stop words
    "a", "an", "the", "and", "or", "but", "if", "while", "with",
    "of", "at", "by", "for", "to", "in", "on", "off", "out", "up",
    "down", "over", "under", "again", "further", "then", "once",
    "here", "there", "when", "where", "why", "how", "all", "any",
    "both", "each", "few", "more", "most", "other", "some", "such",
    "no", "nor", "not", "only", "own", "same", "so", "than", "too",
    "very", "can", "will", "just", "don", "should", "now",

    # Pronouns
    "i", "me", "my", "mine", "you", "your", "yours",
    "he", "him", "his", "she", "her", "they", "them", "their",
    "we", "us", "our", "its", "it",

    # Auxiliary verbs
    "is", "am", "are", "was", "were", "be", "being", "been",
    "do", "does", "did", "have", "has", "had",
    "will", "would", "shall", "should", "can", "could", "may", "might", "must",

    # Informal/slang tokens from dataset
    "im", "idk", "u", "wa", "gon", "na", "tho", "thats", "ur",

    # Single letters/numbers that often aren't meaningful
    "a", "i", "u", "2", "3", "4", "5", "6", "7", "8", "9", "0",

    # extra added to deal with links or that I needed to add manually
    ":", "as", "into", "until", "among", "like", "dont", "from", "doesnt",␣
  ↪"that", "be", "ha", "thi",
}

def word_based_processing(text):
    # stems text and removes common stop words
    # and removes details of https and http links (but keeps the http or https)
    # and removes username tags after the @ (but keeps the @)
    temp_words = []
    for word in text.split(" "):
        if word.startswith("https:"):
            temp_words.append(word[:5])
        elif word.startswith("http:"):
            temp_words.append(word[:4])
        elif word.startswith("@"):
            temp_words.append(word[:1])
        else:
            temp_words.append(word)
    text_links_handled = " ".join(temp_words)
    words = word_tokenize(text_links_handled)
    words_processed = []
    for word in words:
        if isinstance(word, str):
            stem = ps.stem(word)
        # remove common stops and : that lingers
        if stem not in common_stops:
            words_processed.append(stem)
    return " ".join(words_processed)

def filter_for_alphanumeric_and_more(text):
    # filters to only keep a-z, 0-9, @, #, and spaces and :
    return re.sub(r'[^a-z0-9@# :]', '', text)

def preprocess(df):
    new_df = df.copy()
    print("Original:")
    print(new_df.head())
    new_df["text"] = new_df["text"].str.lower()
    print("Lowercase:")
    print(new_df.head())
    new_df["text"] = new_df["text"].apply(filter_for_alphanumeric_and_more)
    print("Filtered:")
    print(new_df.head())
    new_df["text"] = new_df["text"].apply(word_based_processing)
    print("Filtered wordbased:")
    print(new_df.head())
    return new_df
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /home/jiwonjjeong/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to
[nltk_data]     /home/jiwonjjeong/nltk_data…
[nltk_data]   Package punkt_tab is already up-to-date!
```

[95]:
```
# Preprocess test
data = {
    "Id": [1,2,3,4,5],
    "text": ["SOOOO PUMPED FOR ABLAZE ???? @southridgelife",
             "Check these out: http://t.co/rOI2NSmEJJ http://t.co/3Tj8ZjiN21␣
 ↪http://t.co/YDUiXEfIpE http://t.co/LxTjc87KLS #nsfw",
             "Rene Ablaze &amp; Jacinta - Secret 2k13 (Fallen Skies Edit) - Mar␣
 ↪30 2013  https://t.co/7MLMsUzV1Z",
             "@flowri were you marinading it or was it an accident?",
             "105        320 [IR] ICEMOON [AFTERSHOCK] | http://t.co/yNXnvVKCDA␣
↪| @djicemoon | #Dubstep #TrapMusic #DnB #EDM #Dance #Ices Û_ http://t.co/
↪weQPesENku"]
}

df = pd.DataFrame(data)
preprocess(df)
```

```
Original:
   Id                                               text
0   1        SOOOO PUMPED FOR ABLAZE ???? @southridgelife
1   2  Check these out: http://t.co/rOI2NSmEJJ http:/…
2   3  Rene Ablaze &amp; Jacinta - Secret 2k13 (Falle…
3   4  @flowri were you marinading it or was it an ac…
4   5  105\t320 [IR] ICEMOON [AFTERSHOCK] | http://t…
Lowercase:
   Id                                               text
0   1        soooo pumped for ablaze ???? @southridgelife
1   2  check these out: http://t.co/roi2nsmejj http:/…
2   3  rene ablaze &amp; jacinta - secret 2k13 (falle…
3   4  @flowri were you marinading it or was it an ac…
4   5  105\t320 [ir] icemoon [aftershock] | http://t…
Filtered:
   Id                                               text
0   1            soooo pumped for ablaze  @southridgelife
1   2  check these out: http:tcoroi2nsmejj http:tco3t…
2   3  rene ablaze amp jacinta  secret 2k13 fallen sk…
3   4  @flowri were you marinading it or was it an ac…
4   5  105320 ir icemoon aftershock  http:tcoynxnvvkc…
Filtered wordbased:
   Id                                               text
0   1                                 soooo pump ablaz @
1   2               check these http http http http # nsfw
2   3  rene ablaz amp jacinta secret 2k13 fallen sky …
3   4                                    @ marinad accid
4   5  105320 ir icemoon aftershock http @ # dubstep …
```

[95]:
```
   Id                                               text
0   1                                 soooo pump ablaz @
1   2               check these http http http http # nsfw
2   3  rene ablaz amp jacinta secret 2k13 fallen sky …
3   4                                    @ marinad accid
4   5  105320 ir icemoon aftershock http @ # dubstep …
```

[96]:
```
# Preprocess all and save
preprocess(train).to_csv("data-from-NLP-disaster-tweets/preprocessed/train.
↪csv", index=False)
```

```
preprocess(validate).to_csv("data-from-NLP-disaster-tweets/preprocessed/
  ↪validate.csv", index=False)
preprocess(test).to_csv("data-from-NLP-disaster-tweets/preprocessed/test.csv",␣
  ↪index=False)
```

```
Original:
         id            keyword                location  \
1186   1707  bridge%20collapse                     NaN
4071   5789               hail  Carol Stream, Illinois
5461   7789             police                Houston
5787   8257            rioting                     NaN
7445  10656             wounds          Lake Highlands

                                                   text  target
1186  Ashes 2015: Australia Ûªs collapse at Trent Br…       0
4071  GREAT MICHIGAN TECHNIQUE CAMP\nB1G THANKS TO @…       1
5461  CNN: Tennessee movie theater shooting suspect …       1
5787  Still rioting in a couple of hours left until …       1
7445  Crack in the path where I wiped out this morni…       0
Lowercase:
         id            keyword                location  \
1186   1707  bridge%20collapse                     NaN
4071   5789               hail  Carol Stream, Illinois
5461   7789             police                Houston
5787   8257            rioting                     NaN
7445  10656             wounds          Lake Highlands

                                                   text  target
1186  ashes 2015: australia ûªs collapse at trent br…       0
4071  great michigan technique camp\nb1g thanks to @…       1
5461  cnn: tennessee movie theater shooting suspect …       1
5787  still rioting in a couple of hours left until …       1
7445  crack in the path where i wiped out this morni…       0
Filtered:
         id            keyword                location  \
1186   1707  bridge%20collapse                     NaN
4071   5789               hail  Carol Stream, Illinois
5461   7789             police                Houston
5787   8257            rioting                     NaN
7445  10656             wounds          Lake Highlands

                                                   text  target
1186  ashes 2015: australias collapse at trent bridg…       0
4071  great michigan technique campb1g thanks to @bm…       1
5461  cnn: tennessee movie theater shooting suspect …       1
5787  still rioting in a couple of hours left until …       1
7445  crack in the path where i wiped out this morni…       0
Filtered wordbased:
        id           keyword               location  \
1186   1707  bridge%20collapse                    NaN
4071   5789              hail  Carol Stream, Illinois
5461   7789            police                Houston
5787   8257           rioting                    NaN
7445  10656            wounds          Lake Highlands

                                                   text  target
1186  ash 2015 australia collaps trent bridg worst h…       0
4071  great michigan techniqu campb1g thank @ @ @ # …       1
5461  cnn tennesse movi theater shoot suspect kill p…       1
5787                 still riot coupl hour left class       1
7445  crack path wipe morn dure beach run surfac wou…       0
Original:
        id      keyword              location  \
2644  3796  destruction                   NaN
2227  3185       deluge                   NaN
5448  7769       police                    UK
132    191   aftershock                   NaN
6845  9810       trauma  Montgomery County, MD

                                                   text  target
2644  So you have a new weapon that can cause un-ima…       1
2227  The f$&amp;@ing things I do for #GISHWHES Just…       0
5448  DT @georgegalloway: RT @Galloway4Mayor: ÛÏThe…       1
132   Aftershock back to school kick off was great. …       0
6845  in response to trauma Children of Addicts deve…       0
Lowercase:
        id      keyword              location  \
2644  3796  destruction                   NaN
2227  3185       deluge                   NaN
5448  7769       police                    UK
132    191   aftershock                   NaN
6845  9810       trauma  Montgomery County, MD

                                                   text  target
2644  so you have a new weapon that can cause un-ima…       1
2227  the f$&amp;@ing things i do for #gishwhes just…       0
5448  dt @georgegalloway: rt @galloway4mayor: ûïthe…       1
132   aftershock back to school kick off was great. …       0
6845  in response to trauma children of addicts deve…       0
Filtered:
        id      keyword              location  \
2644  3796  destruction                   NaN
2227  3185       deluge                   NaN
5448  7769       police                    UK
132    191   aftershock                   NaN
6845  9810       trauma  Montgomery County, MD
```

```
                                                   text  target
2644  so you have a new weapon that can cause unimag…       1
2227  the famp@ing things i do for #gishwhes just go…       0
5448  dt @georgegalloway: rt @galloway4mayor: the co…       1
132   aftershock back to school kick off was great i…       0
6845  in response to trauma children of addicts deve…       0
Filtered wordbased:
        id      keyword              location  \
2644  3796  destruction                   NaN
2227  3185       deluge                   NaN
5448  7769       police                    UK
132    191   aftershock                   NaN
6845  9810       trauma  Montgomery County, MD

                                                   text  target
2644                  new weapon caus unimagin destruct       1
2227  famp @ ing thing # gishwh got soak delug go pa…       0
5448  dt @ rt @ col polic catch pickpocket liverpool…       1
132   aftershock back school kick great want thank e…       0
6845  respons trauma children addict develop defens …       0
Original:
   id keyword location                                             text
0   0     NaN      NaN                 Just happened a terrible car crash
1   2     NaN      NaN  Heard about #earthquake is different cities, s…
2   3     NaN      NaN  there is a forest fire at spot pond, geese are…
3   9     NaN      NaN            Apocalypse lighting. #Spokane #wildfires
4  11     NaN      NaN      Typhoon Soudelor kills 28 in China and Taiwan
Lowercase:
   id keyword location                                             text
0   0     NaN      NaN                 just happened a terrible car crash
1   2     NaN      NaN  heard about #earthquake is different cities, s…
2   3     NaN      NaN  there is a forest fire at spot pond, geese are…
3   9     NaN      NaN            apocalypse lighting. #spokane #wildfires
4  11     NaN      NaN      typhoon soudelor kills 28 in china and taiwan
Filtered:
   id keyword location                                             text
0   0     NaN      NaN                 just happened a terrible car crash
1   2     NaN      NaN  heard about #earthquake is different cities st…
2   3     NaN      NaN  there is a forest fire at spot pond geese are …
3   9     NaN      NaN             apocalypse lighting #spokane #wildfires
4  11     NaN      NaN      typhoon soudelor kills 28 in china and taiwan
Filtered wordbased:
   id keyword location                                             text
0   0     NaN      NaN                            happen terribl car crash
1   2     NaN      NaN  heard about # earthquak differ citi stay safe …
2   3     NaN      NaN  forest fire spot pond gees flee across street …
3   9     NaN      NaN                 apocalyps light # spokan # wildfir
```

```
4   11     NaN      NaN              typhoon soudelor kill 28 china taiwan
```

[97]:
```
# SUMMARY OF PREPROCESSING
# Lowercase all letters
# Only keep alphanumerics, # and @
# Keep the http or https part of links, get rid of the rest of the link
# Keep the @ part of @usernames, get rid of the actual username
# Keep the # of #topics, keep both topic and # but separate them
# Lemmatize all words
# Remove common stop words (and, or, the ..)
```

1.4 Bag of words model

I built a bag of words with different threshold of M. This means that a word needs to appear in at least M different tweets to be included in the bag of words. A bag of words model uses these d words that meet the threshold criteria to create features as a d-dimensional vector. Each element of this vector would be 0 or 1, depending on if that feature did not have that word or not. For example, for 3 words, there would be 8 different features (0,0,0), (0,0,1), (0,1,0) … (1,1,1). And a specific realization (a specific tweet) would be converted to one bag of word vector and match only one feature.

[36]:
```
# Explore word distribution for bag of words

# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "create a dictionary of unique words in my df of text with values as␣
 ↪the count they appear"

import pandas as pd
from collections import Counter

train = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/train.csv")
train_text = train["text"]

# Combine all text rows into one string
all_text = " ".join(train_text.astype(str))

# Split into words (basic split, you can improve with regex if needed)
words = all_text.split()

# Count word frequencies
word_counts = Counter(words)

# Convert to dict if you want a pure dictionary
word_dict = dict(word_counts)

frequency_dict = {}
for freq in word_dict.values():
    if (freq in frequency_dict):
        frequency_dict[freq] += 1
    else:
        frequency_dict[freq] = 1

print(frequency_dict)
print(frequency_dict[1])
print(frequency_dict[2])
print(frequency_dict[3])
print(frequency_dict[4])
print(frequency_dict[5])
print(frequency_dict[6])
print(frequency_dict[7])
```

```
{13: 50, 42: 7, 17: 38, 77: 4, 9: 111, 24: 20, 14: 55, 19: 34, 8: 132, 3: 636,
15: 41, 3245: 1, 44: 9, 4: 391, 1: 6084, 46: 10, 1946: 1, 2404: 1, 5: 274, 39:
13, 10: 82, 25: 25, 26: 16, 116: 1, 86: 2, 91: 1, 58: 3, 20: 36, 29: 20, 21: 31,
51: 1, 53: 4, 2: 1310, 12: 70, 11: 77, 28: 19, 36: 19, 35: 11, 34: 8, 68: 3, 18:
30, 129: 1, 48: 7, 6: 196, 119: 1, 70: 3, 50: 7, 30: 16, 60: 3, 186: 1, 71: 1,
151: 1, 87: 2, 142: 1, 73: 3, 69: 2, 7: 147, 37: 8, 65: 5, 23: 38, 110: 2, 90:
1, 62: 4, 74: 3, 38: 6, 16: 33, 40: 8, 130: 1, 81: 3, 211: 1, 22: 24, 107: 1,
32: 12, 92: 1, 82: 2, 164: 1, 93: 1, 33: 12, 254: 1, 78: 1, 56: 2, 64: 1, 104:
1, 88: 4, 144: 1, 105: 1, 54: 3, 52: 3, 220: 1, 89: 1, 27: 20, 102: 1, 66: 5,
97: 1, 72: 1, 67: 3, 114: 2, 150: 2, 80: 1, 158: 1, 99: 1, 59: 4, 112: 2, 75: 2,
47: 4, 49: 6, 152: 1, 167: 1, 98: 1, 43: 3, 41: 5, 123: 1, 31: 7, 94: 1, 79: 1,
103: 1, 45: 1}
6084
1310
636
391
274
196
147
```

```
[36]: 0
```

As seen with the 3 word example, it is very easy for the dimensionality of the features to grow, so it was important to choose a minimum threshold M that was as high as possible without losing model predictive power. To choose M, I counted the frequency of each word and discovered that 6084 words only appeared once, 1310 words appeared only twice, 636 words only appeared three times, and 391 words only appeared four times. There on, the frequency only continues to decrease. As seen, setting M=2 would result in a significant removal of 6084 words. This is a good idea as well, since tokens that only appear once are probabilistically unlikely to represent the data generating distribution well. Removing these single-frequency words reduces likelihood of overfitting. I also decided to remove words that only appear twice for the same reason and for even more memory efficiency. However, deciding to choose M=3,4,5,6, or 7 was unclear, so I decided to create models on all these M values to evaluate the models on a later step.

[37]:
```
# Check if very high frequency words are missed stop words
for word, freq in word_dict.items():
    if freq > 500:
        print(word)
```

```
http
@
#
```

[43]:
```
# Choose words for bag of words
from sklearn.feature_extraction.text import CountVectorizer
# I want to choose all words with frequency = 3+, 4+, 5+, 6+, 7+ (5 different)
vectorizer_3 = CountVectorizer(binary=True, min_df=3)
vectorizer_4 = CountVectorizer(binary=True, min_df=4)
vectorizer_5 = CountVectorizer(binary=True, min_df=5)
vectorizer_6 = CountVectorizer(binary=True, min_df=6)
vectorizer_7 = CountVectorizer(binary=True, min_df=7)
train = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/train.csv")
validate = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/validate.
 ↪csv")
train_text = train["text"]
validate_text = validate["text"]
validate_text.head()
```

[43]:
```
0                     new weapon caus unimagin destruct
1    famp @ ing thing # gishwh got soak delug go pa…
2    dt @ rt @ col polic catch pickpocket liverpool…
3    aftershock back school kick great want thank e…
4    respons trauma children addict develop defens …
Name: text, dtype: object
```

[137]:
```
# Define function to create features for training and validation sets
def extract_features(vectorizer, train, validate):
    train_feats = vectorizer.fit_transform(train)
    words_selected = vectorizer.get_feature_names_out()
    vectorizer_val = CountVectorizer(vocabulary = words_selected)
    validate_feats = vectorizer_val.fit_transform(validate)
    return train_feats, validate_feats, words_selected
```

[138]:
```
# Run and store generated features from bag of words
feats = {}

feats[3] = extract_features(vectorizer_3, train_text, validate_text)
feats[4] = extract_features(vectorizer_4, train_text, validate_text)
feats[5] = extract_features(vectorizer_5, train_text, validate_text)
feats[6] = extract_features(vectorizer_6, train_text, validate_text)
feats[7] = extract_features(vectorizer_7, train_text, validate_text)
print(feats[3][0].shape)
print(feats[3][1].shape)
print(feats[7][0].shape)
```

```
(5329, 2884)
(2284, 2884)
(5329, 1403)
```

1.5 Logistic regression

1.5.1 Regression without regularization

A logistic regression without regularization was performed on each of the 5 feature sets created by M=3,4,5,6,7.

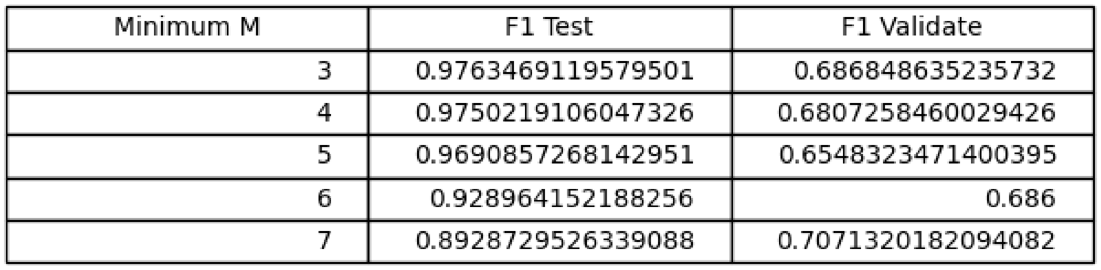
[79]:
```
# Define logistic regression without regularization
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score
train_label = train["target"]
validate_label = validate["target"]

def logistic_regression_noreg(X_train, X_validate, Y_train=train_label,␣
 ↪Y_validate=validate_label):
    clf = LogisticRegression(random_state=0, penalty=None, max_iter=5000).
 ↪fit(X_train, Y_train)
    Y_pred = clf.predict(X_validate)
    Y_pred_train = clf.predict(X_train)
    score_test = f1_score(Y_train, Y_pred_train)
    score_validate = f1_score(Y_validate, Y_pred)
    print(f"Training F1: {score_test}, Validate F1: {score_validate}")
    return score_test, score_validate
```

For all feature sets, the F1 score on the training set was very high (>0.89), but the F1 scores on the validation set was not very high (< 0.71). This is a sign that the models have good training loss, but poor generalization (also known as overfitting).

[151]:
```
# Actually run regression
data = []
for min_df in feats.keys():
    test, val = logistic_regression_noreg(feats[min_df][0], feats[min_df][1])
    data.append((min_df, test, val))

# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how to create a table from a list of tuples where I want the list␣
 ↪[(a,1),(b,2)] to create a table with columns of letters and columns of␣
 ↪numbers"
fig, ax = plt.subplots()

# Hide axes
ax.axis('off')
```

```
# Create table
table = ax.table(cellText=data, colLabels=["Minimum M", "F1 Test", "F1␣
↪Validate"], loc='center')
table.auto_set_font_size(True)
table.scale(1.2, 1.4)

plt.show()
```

```
Training F1: 0.9763469119579501, Validate F1: 0.686848635235732
Training F1: 0.9750219106047326, Validate F1: 0.6807258460029426
Training F1: 0.9690857268142951, Validate F1: 0.6548323471400395
Training F1: 0.928964152188256, Validate F1: 0.686
Training F1: 0.8928729526339088, Validate F1: 0.7071320182094082
```



Minimum M	F1 Test	F1 Validate
3	0.9763469119579501	0.686848635235732
4	0.9750219106047326	0.6807258460029426
5	0.9690857268142951	0.6548323471400395
6	0.928964152188256	0.686
7	0.8928729526339088	0.7071320182094082

1.5.2 Regression with L1 regularization

A logistic regression with L1 regularization was performed on each of the 5 feature sets created by M=3,4,5,6,7.

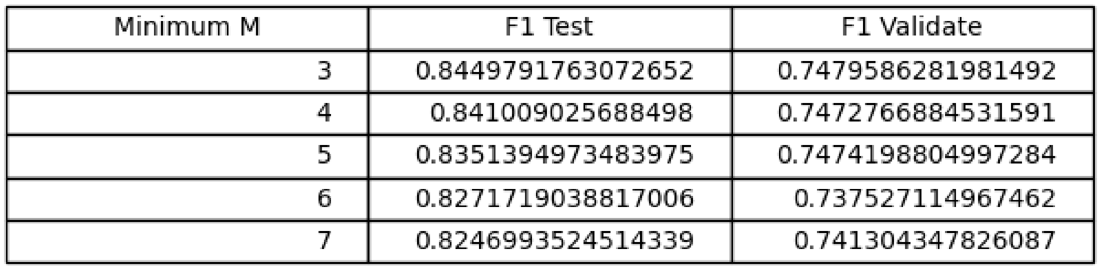
[171]:
```
# Define logistic regression with L1 regularization
def logistic_regression_l1(X_train, X_validate, Y_train=train_label,␣
↪Y_validate=validate_label, C=1):
    clf = LogisticRegression(random_state=0, penalty="l1", solver="saga",␣
↪max_iter=2000, C=C).fit(X_train, Y_train)
```

```
    Y_pred = clf.predict(X_validate)
    Y_pred_train = clf.predict(X_train)
    score_test = f1_score(Y_train, Y_pred_train)
    score_validate = f1_score(Y_validate, Y_pred)
    print(f"Training F1: {score_test}, Validate F1: {score_validate}")
    weights = clf.coef_
    return score_test, score_validate, weights
```

Once again, the F1 score on the testing set is higher than the scores on the validation set, but the difference is now much less. Specifically, we see the testing F1 scores drop and validation F1 scores rise. Thus, with regularization, the overfitting has been reduced.

[154]:
```
# Actually run regression
data = []
weights = []
for min_df in feats.keys():
    test, val, weight = logistic_regression_l1(feats[min_df][0],␣
 ↪feats[min_df][1])
    data.append((min_df, test, val))
    weights.append(weight)

# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how to create a table from a list of tuples where I want the list␣
 ↪[(a,1),(b,2)] to create a table with columns of letters and columns of␣
 ↪numbers"
fig, ax = plt.subplots()

# Hide axes
ax.axis('off')

# Create table
table = ax.table(cellText=data, colLabels=["Minimum M", "F1 Test", "F1␣
 ↪Validate"], loc='center')
table.auto_set_font_size(True)
table.scale(1.2, 1.4)

plt.show()
```

```
Training F1: 0.8449791763072652, Validate F1: 0.7479586281981492
Training F1: 0.841009025688498, Validate F1: 0.7472766884531591
Training F1: 0.8351394973483975, Validate F1: 0.7474198804997284
Training F1: 0.8271719038817006, Validate F1: 0.737527114967462
Training F1: 0.8246993524514339, Validate F1: 0.741304347826087
```



Minimum M	F1 Test	F1 Validate
3	0.8449791763072652	0.7479586281981492
4	0.841009025688498	0.7472766884531591
5	0.8351394973483975	0.7474198804997284
6	0.8271719038817006	0.737527114967462
7	0.8246993524514339	0.741304347826087

1.5.3 Regression with L2 regularization

A logistic regression with L2 regularization was performed on each of the 5 feature sets created by M=3,4,5,6,7.

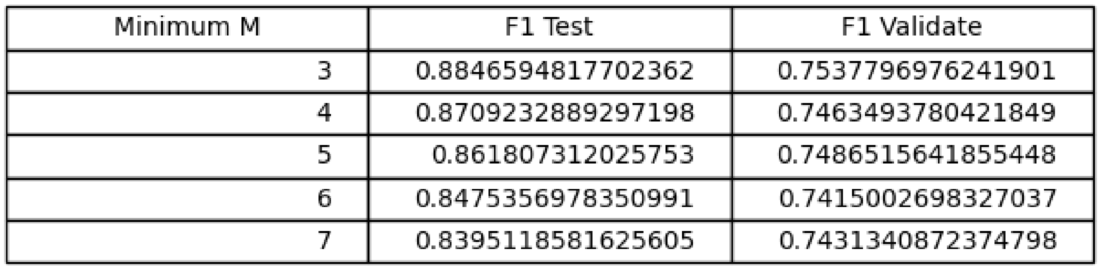
```
[170]: # Define logistic regression with L2 regularization
       def logistic_regression_l2(X_train, X_validate, Y_train=train_label,␣
        ↪Y_validate=validate_label, C=1):
           clf = LogisticRegression(random_state=0, penalty="l2", max_iter=5000, C=C).
        ↪fit(X_train, Y_train)
           Y_pred = clf.predict(X_validate)
           Y_pred_train = clf.predict(X_train)
           score_test = f1_score(Y_train, Y_pred_train)
           score_validate = f1_score(Y_validate, Y_pred)
           print(f"Training F1: {score_test}, Validate F1: {score_validate}")
           return score_test, score_validate
```

Similar to the L1 regularization scores, there is less overfitting compared to the no regularization scores. Additionally the scores for the L2 regularizations are better than the scores for the L1 regularization, showing that the L1 regularization underfits slight more than L2 regularization.

```
[156]: # Actually run regression
       data = []
       for min_df in feats.keys():
           test, val = logistic_regression_l2(feats[min_df][0], feats[min_df][1])
```

```
    data.append((min_df, test, val))

# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "how to create a table from a list of tuples where I want the list␣
 ↪[(a,1),(b,2)] to create a table with columns of letters and columns of␣
 ↪numbers"
fig, ax = plt.subplots()

# Hide axes
ax.axis('off')

# Create table
table = ax.table(cellText=data, colLabels=["Minimum M", "F1 Test", "F1␣
 ↪Validate"], loc='center')
table.auto_set_font_size(True)
table.scale(1.2, 1.4)

plt.show()
```

```
Training F1: 0.8846594817702362, Validate F1: 0.7537796976241901
Training F1: 0.8709232889297198, Validate F1: 0.7463493780421849
Training F1: 0.861807312025753, Validate F1: 0.7486515641855448
Training F1: 0.8475356978350991, Validate F1: 0.7415002698327037
Training F1: 0.8395118581625605, Validate F1: 0.7431340872374798
```



Minimum M	F1 Test	F1 Validate
3	0.8846594817702362	0.7537796976241901
4	0.8709232889297198	0.7463493780421849
5	0.861807312025753	0.7486515641855448
6	0.8475356978350991	0.7415002698327037
7	0.8395118581625605	0.7431340872374798

1.5.4 Discussion of regularization

The training set score was best with no regularization. The validation set score was best on the L2 regularization. As discussed, overfitting was observed on the no regularization models (high training score but low validation score). For all feature sets, the F1 score on the testing set was very high (>0.89), but the F1 scores on the validation set was not very high (< 0.71).

Adding either L1 or L2 regularization helped reduce overfitting. This was observed in both the L1 and L2 regularization as an increase in the validation (generalization) score from around 0.68 to around 0.74. As a byproduct, the training score also dips a little.

To investigate the effect of regularization further, I tested various regularization coefficient hyperparameters on each of the feature sets. A high coefficient means less regularization. As expected, adding more and more regularization increases the validation F1 score while decreasing training F1 score, showing a reduction in overfitting. However at some point with very high regularization, the F1 validation score also begins to drop and the F1 training score is low. This is a sign of underfitting due to a too-extreme regularization constant. Surprisingly the optimal regularization hyperparameter was 1, the default. The best model of all of these hyperparameters was for the M=3 feature set using L2 regularization with C=1. The F1 validation score was 0.754.

[172]:
```
# Now combine testing for both minimum M and regularization constant
data = [] # is (minimum M, regularization type, reg constant, F1 test, F1␣
 ↪validate)
reg_constants = [100,10,1,0.1,0.01]
for min_df in feats.keys():
    test, val = logistic_regression_noreg(feats[min_df][0], feats[min_df][1])
    data.append((min_df, "None", "N/A", test, val))
    for reg in reg_constants:
        test, val, weight = logistic_regression_l1(feats[min_df][0],␣
 ↪feats[min_df][1], C=reg)
        data.append((min_df, "L1", reg, test, val))
    for reg in reg_constants:
        test, val = logistic_regression_l2(feats[min_df][0], feats[min_df][1],␣
 ↪C=reg)
        data.append((min_df, "L2", reg, test, val))
```

```
Training F1: 0.9763469119579501, Validate F1: 0.686848635235732
```

```
/home/jiwonjjeong/cornell-tech/CT-applied-machine-
learning/venv/lib/python3.12/site-packages/sklearn/linear_model/_sag.py:348:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

```
Training F1: 0.9763469119579501, Validate F1: 0.6783391695847923
Training F1: 0.9671878440872055, Validate F1: 0.7024586051179127
Training F1: 0.8449791763072652, Validate F1: 0.7479586281981492
Training F1: 0.635489043292357, Validate F1: 0.6227544910179641
Training F1: 0.5946053153510512, Validate F1: 0.5968194574368568
Training F1: 0.9719298245614035, Validate F1: 0.687624750499002
Training F1: 0.9506309497454063, Validate F1: 0.72
Training F1: 0.8846594817702362, Validate F1: 0.7537796976241901
Training F1: 0.7890794877989853, Validate F1: 0.7407407407407407
Training F1: 0.6855952070851784, Validate F1: 0.6701631701631702
Training F1: 0.9750219106047326, Validate F1: 0.6807258460029426
```

```
/home/jiwonjjeong/cornell-tech/CT-applied-machine-
learning/venv/lib/python3.12/site-packages/sklearn/linear_model/_sag.py:348:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

```
Training F1: 0.9723562966213252, Validate F1: 0.6760140210315473
Training F1: 0.9555604687154543, Validate F1: 0.7060020345879959
Training F1: 0.841009025688498, Validate F1: 0.7472766884531591
Training F1: 0.635489043292357, Validate F1: 0.6227544910179641
Training F1: 0.5946053153510512, Validate F1: 0.5968194574368568
Training F1: 0.965941551307405, Validate F1: 0.6919575113808801
Training F1: 0.931934835974113, Validate F1: 0.7182952182952183
Training F1: 0.8709232889297198, Validate F1: 0.7463493780421849
Training F1: 0.7836398838334947, Validate F1: 0.7407407407407407
Training F1: 0.6838810641627543, Validate F1: 0.6666666666666666
Training F1: 0.9690857268142951, Validate F1: 0.6548323471400395
```

```
/home/jiwonjjeong/cornell-tech/CT-applied-machine-
learning/venv/lib/python3.12/site-packages/sklearn/linear_model/_sag.py:348:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

```
Training F1: 0.9588377723970944, Validate F1: 0.6780684104627767
Training F1: 0.934984520123839, Validate F1: 0.6998982706002035
Training F1: 0.8351394973483975, Validate F1: 0.7474198804997284
Training F1: 0.635489043292357, Validate F1: 0.6227544910179641
Training F1: 0.5946053153510512, Validate F1: 0.5968194574368568
Training F1: 0.9464875578066505, Validate F1: 0.6950354609929078
Training F1: 0.9127426913635349, Validate F1: 0.7217030114226376
Training F1: 0.861807312025753, Validate F1: 0.7486515641855448
Training F1: 0.7786666666666666, Validate F1: 0.7374929735806633
Training F1: 0.6809287764153404, Validate F1: 0.6670547147846333
Training F1: 0.928964152188256, Validate F1: 0.686
```

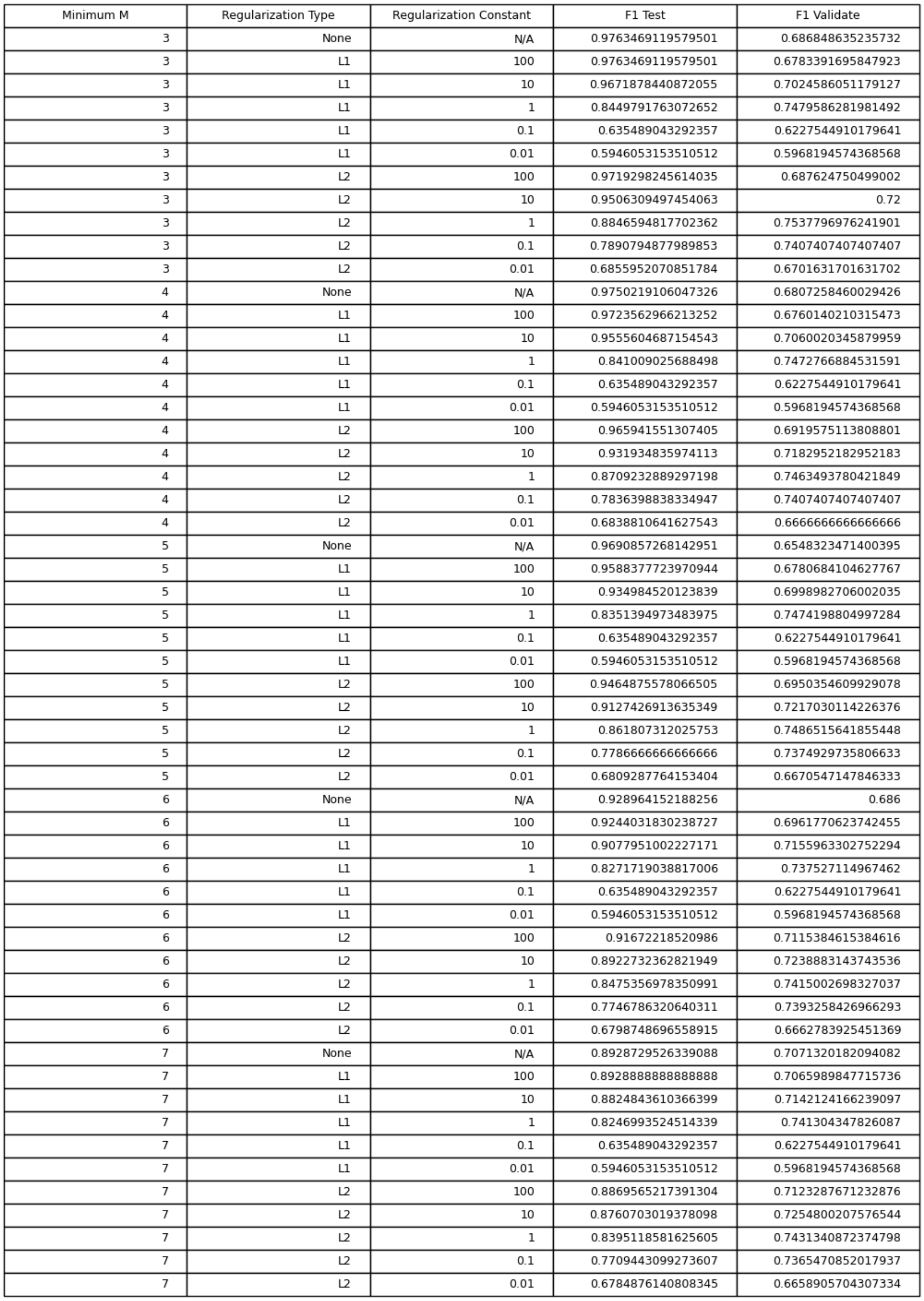
```
/home/jiwonjjeong/cornell-tech/CT-applied-machine-
learning/venv/lib/python3.12/site-packages/sklearn/linear_model/_sag.py:348:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

```
Training F1: 0.9244031830238727, Validate F1: 0.6961770623742455
Training F1: 0.9077951002227171, Validate F1: 0.7155963302752294
Training F1: 0.8271719038817006, Validate F1: 0.737527114967462
Training F1: 0.635489043292357, Validate F1: 0.6227544910179641
Training F1: 0.5946053153510512, Validate F1: 0.5968194574368568
Training F1: 0.91672218520986, Validate F1: 0.7115384615384616
Training F1: 0.8922732362821949, Validate F1: 0.7238883143743536
Training F1: 0.8475356978350991, Validate F1: 0.7415002698327037
Training F1: 0.7746786320640311, Validate F1: 0.7393258426966293
Training F1: 0.6798748696558915, Validate F1: 0.6662783925451369
Training F1: 0.8928729526339088, Validate F1: 0.7071320182094082
```

```
/home/jiwonjjeong/cornell-tech/CT-applied-machine-
learning/venv/lib/python3.12/site-packages/sklearn/linear_model/_sag.py:348:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  warnings.warn(
```

```
Training F1: 0.8928888888888888, Validate F1: 0.7065989847715736
Training F1: 0.8824843610366399, Validate F1: 0.7142124166239097
Training F1: 0.8246993524514339, Validate F1: 0.741304347826087
Training F1: 0.635489043292357, Validate F1: 0.6227544910179641
Training F1: 0.5946053153510512, Validate F1: 0.5968194574368568
Training F1: 0.8869565217391304, Validate F1: 0.7123287671232876
Training F1: 0.8760703019378098, Validate F1: 0.7254800207576544
Training F1: 0.8395118581625605, Validate F1: 0.7431340872374798
Training F1: 0.7709443099273607, Validate F1: 0.7365470852017937
Training F1: 0.6784876140808345, Validate F1: 0.6658905704307334
```

[185]:
```python
# Graph all results
import numpy as np

arr = np.array(data)  # shape (n_rows, n_columns)
labels = ["Minimum M", "Regularization Type", "Regularization Constant", "F1␣
↪Test", "F1 Validate"]

fig, ax = plt.subplots()

ax.axis('off')  # hide axes

# Create table
table = ax.table(
    cellText=data,
    colLabels=labels,
    loc='center'
)

table.auto_set_font_size(True)
table.scale(2.0, 1.5)  # adjust cell sizes

plt.show()
```



Minimum M	Regularization Type	Regularization Constant	F1 Test	F1 Validate
3	None	N/A	0.9763469119579501	0.686848635235732
3	L1	100	0.9763469119579501	0.6783391695847923
3	L1	10	0.9671878440872055	0.7024586051179127
3	L1	1	0.8449791763072652	0.7479586281981492
3	L1	0.1	0.635489043292357	0.6227544910179641
3	L1	0.01	0.5946053153510512	0.5968194574368568
3	L2	100	0.9719298245614035	0.687624750499002
3	L2	10	0.9506309497454063	0.72
3	L2	1	0.8846594817702362	0.7537796976241901
3	L2	0.1	0.7890794877989853	0.7407407407407407
3	L2	0.01	0.6855952070851784	0.6701631701631702
4	None	N/A	0.9750219106047326	0.6807258460029426
4	L1	100	0.9723562966213252	0.6760140210315473
4	L1	10	0.9555604687154543	0.7060020345879959
4	L1	1	0.841009025688498	0.7472766884531591
4	L1	0.1	0.635489043292357	0.6227544910179641
4	L1	0.01	0.5946053153510512	0.5968194574368568
4	L2	100	0.965941551307405	0.6919575113808801
4	L2	10	0.931934835974113	0.7182952182952183
4	L2	1	0.8709232889297198	0.7463493780421849
4	L2	0.1	0.7836398838334947	0.7407407407407407
4	L2	0.01	0.6838810641627543	0.6666666666666666
5	None	N/A	0.9690857268142951	0.6548323471400395
5	L1	100	0.9588377723970944	0.6780684104627767
5	L1	10	0.934984520123839	0.6998982706002035
5	L1	1	0.8351394973483975	0.7474198804997284
5	L1	0.1	0.635489043292357	0.6227544910179641
5	L1	0.01	0.5946053153510512	0.5968194574368568
5	L2	100	0.9464875578066505	0.6950354609929078
5	L2	10	0.9127426913635349	0.7217030114226376
5	L2	1	0.861807312025753	0.7486515641855448
5	L2	0.1	0.7786666666666666	0.7374929735806633
5	L2	0.01	0.6809287764153404	0.6670547147846333
6	None	N/A	0.928964152188256	0.686
6	L1	100	0.9244031830238727	0.6961770623742455
6	L1	10	0.9077951002227171	0.7155963302752294
6	L1	1	0.8271719038817006	0.737527114967462
6	L1	0.1	0.635489043292357	0.6227544910179641
6	L1	0.01	0.5946053153510512	0.5968194574368568
6	L2	100	0.91672218520986	0.7115384615384616
6	L2	10	0.8922732362821949	0.7238883143743536
6	L2	1	0.8475356978350991	0.7415002698327037
6	L2	0.1	0.7746786320640311	0.7393258426966293
6	L2	0.01	0.6798748696558915	0.6662783925451369
7	None	N/A	0.8928729526339088	0.7071320182094082
7	L1	100	0.8928888888888888	0.7065989847715736
7	L1	10	0.8824843610366399	0.7142124166239097
7	L1	1	0.8246993524514339	0.741304347826087
7	L1	0.1	0.635489043292357	0.6227544910179641
7	L1	0.01	0.5946053153510512	0.5968194574368568
7	L2	100	0.8869565217391304	0.7123287671232876
7	L2	10	0.8760703019378098	0.7254800207576544
7	L2	1	0.8395118581625605	0.7431340872374798
7	L2	0.1	0.7709443099273607	0.7365470852017937
7	L2	0.01	0.6784876140808345	0.6658905704307334

[186]:
```
# Get highest validation score model
max_model = max(data, key=lambda x: x[-1])
print(max_model)
```

```
(3, 'L2', 1, 0.8846594817702362, 0.7537796976241901)
```

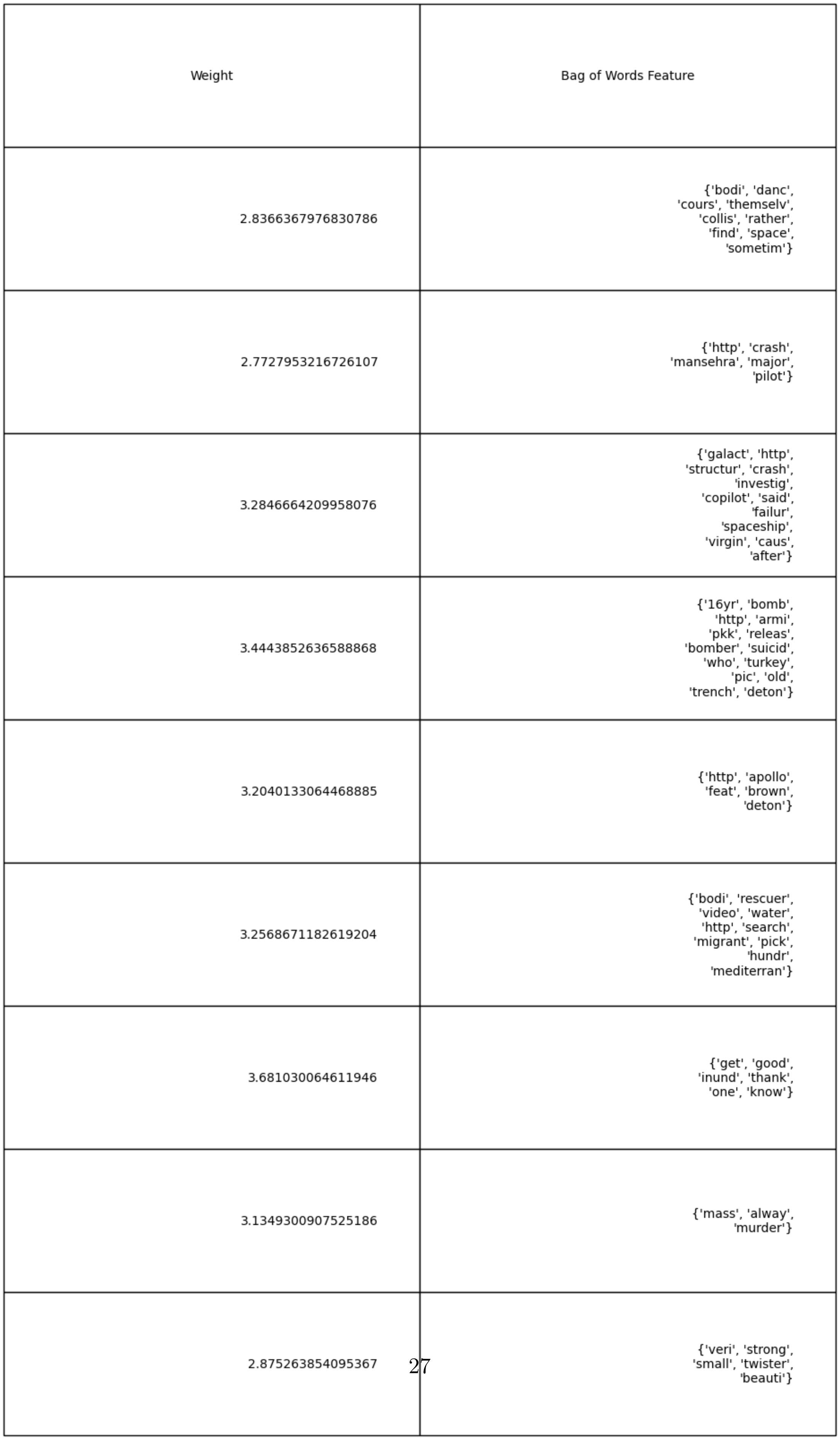
1.5.5 Inspecting weights of L1 regularization

I observed the weights of the best L1 regularization model. As expected with the sparsity of L1, the weight vector had a lot of 0 values, meaning many features had zero effect on a positive classification. I investigated the top few weights and decomposed the bag of words feature to get what words are in that feature. Some of these words are expected to be correlated with disaster tweets like “crash”, “bomb”, and “deton”. However, it is important to keep in mind that the bag of words approach does not treat each word independently. So “crash” is only a strong indicator of a disaster tweet when together with “http”, “mansehra”, “major” and “pilot”.

[169]:
```
# Inspect weights from L1 regularization
# Use same M and C of the best (L2) model
import textwrap

print(weights[0][0])
highest_weights = []
words = []
for i, weight in enumerate(weights[0][0]): # loop and get highest weights
    if weight > 2.7:
        feat = feats[3][0][i] # this gives us feature (0,0,0,1,0,0...)
        array = feat.toarray()
        highest_weights.append(weight)
        temp_word = set()
        for i, val in enumerate(array[0]): # loop through feature for words
            if val == 1:
                # print(feats[3][2][i])
                temp_word.add(feats[3][2][i])
        words.append(temp_word)

# Adapted from code generated by ChatGPT (OpenAI, 2025)
# Prompt: "i have two lists of data that i want to plot in matplot lib table"

max_width = 20  # max chars per line
wrapped_words = ["\n".join(textwrap.wrap(str(word), max_width)) for word in␣
 ↪words]

data = list(zip(highest_weights, wrapped_words))
columns = ["Weight", "Bag of Words Feature"]

fig, ax = plt.subplots(figsize=(10, len(data) * 0.4))  # height scales with rows
ax.axis('off')
```

```python
table = ax.table(cellText=data, colLabels=columns, loc='center')

# Increase font size
table.auto_set_font_size(False)
table.set_fontsize(10)

# Scale the table so row heights match wrapped text
row_height_factor = max(len(w.split("\n")) for w in wrapped_words)  # max lines␣
↪in any cell
table.scale(1.2, 1.2 * row_height_factor)  # increase row height proportionally

plt.show()
```

```
[0.         0.99080227 0.         … 0.         0.         0.        ]
```



Weight	Bag of Words Feature
2.8366367976830786	{'bodi', 'danc', 'cours', 'themselv', 'collis', 'rather', 'find', 'space', 'sometim'}
2.7727953216726107	{'http', 'crash', 'mansehra', 'major', 'pilot'}
3.2846664209958076	{'galact', 'http', 'structur', 'crash', 'investig', 'copilot', 'said', 'failur', 'spaceship', 'virgin', 'caus', 'after'}
3.4443852636588868	{'16yr', 'bomb', 'http', 'armi', 'pkk', 'releas', 'bomber', 'suicid', 'who', 'turkey', 'pic', 'old', 'trench', 'deton'}
3.2040133064468885	{'http', 'apollo', 'feat', 'brown', 'deton'}
3.2568671182619204	{'bodi', 'rescuer', 'video', 'water', 'http', 'search', 'migrant', 'pick', 'hundr', 'mediterran'}
3.681030064611946	{'get', 'good', 'inund', 'thank', 'one', 'know'}
3.1349300907525186	{'mass', 'alway', 'murder'}
2.875263854095367	{'veri', 'strong', 'small', 'twister', 'beauti'}

1.6 N-gram model

I trained a model using the 2-gram model as well.

[223]:
```
# N-gram model
# This time only do highest M=3,4,5
# because we saw how M=3 was best for the N=1 model

from sklearn.feature_extraction.text import CountVectorizer
# I want to choose all words with frequency = 3+, 4+, 5+
vectorizer_3_N2 = CountVectorizer(binary=True, min_df=3, ngram_range=(2,2))
vectorizer_4_N2 = CountVectorizer(binary=True, min_df=4, ngram_range=(2,2))
vectorizer_5_N2 = CountVectorizer(binary=True, min_df=5, ngram_range=(2,2))
train = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/train.csv")
validate = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/validate.
 ↪csv")
train_text = train["text"]
validate_text = validate["text"]
validate_text.head()
```

[223]:
```
0                     new weapon caus unimagin destruct
1    famp @ ing thing # gishwh got soak delug go pa…
2    dt @ rt @ col polic catch pickpocket liverpool…
3    aftershock back school kick great want thank e…
4    respons trauma children addict develop defens …
Name: text, dtype: object
```

For threshold M=3, there are 1779 2-grams. These only include the two word tokens. Combined with the 1-grams, the threshold M=3 has 4663 grams. For M=4, there are 1101 2-grams and 3353 total 1- and 2-grams. For M=5, there are 762 2-grams and 2632 total 1- and 2-grams. As seen, there is a choice to train only on the 2-grams or also include the 1-grams. Thus I created another feature set to evaluate. With 3 different threshold values and two gram-related chocies for each, there are 6 feature sets to test. However, due to memory constraints I decided to only test L2 regularizations. L1 regularizations required much more computational power (likely due to non-differentiability of absolute values). This choice likely is not significant, because L2 regularization models consistently performed better than L1 regularizations on the bag of word models.

[224]:
```
# Determine number of 2-grams
feats_N = {}

feats_N[3] = extract_features(vectorizer_3_N2, train_text, validate_text)
feats_N[4] = extract_features(vectorizer_4_N2, train_text, validate_text)
feats_N[5] = extract_features(vectorizer_5_N2, train_text, validate_text)
print(feats_N[3][0].shape)
print(feats_N[4][0].shape)
print(feats_N[5][0].shape)
```

```
(5329, 1779)
(5329, 1101)
(5329, 762)
```

Some examples of 2-grams were “abandon aircraft”, “abbswinston zionist”, “abc news”, “about cabl”, “about trap”, “access secret”, “accid expert”, “accid http”, “accid indian”, and “accid man”. Based on the feature set study on the bag of words model, the threshold M=3 was the best feature set for the F1 validation score. Thus, I expected a similar result, but I still chose to test a few different feature sets with threshold M=3,4,5 to confirm.

[225]:
```
# Print out 10 2-grams in the M=3 set
for i in range(10):
    print(feats_N[3][2][i+50])
```

```
abandon aircraft
abbswinston zionist
abc news
about cabl
about trap
access secret
accid expert
accid http
accid indian
accid man
```

[226]:
```
# Get both 1 and 2-grams
vectorizer_3_N2 = CountVectorizer(binary=True, min_df=3, ngram_range=(1,2))
vectorizer_4_N2 = CountVectorizer(binary=True, min_df=4, ngram_range=(1,2))
vectorizer_5_N2 = CountVectorizer(binary=True, min_df=5, ngram_range=(1,2))
feats_N_12 = {}

feats_N_12[3] = extract_features(vectorizer_3_N2, train_text, validate_text)
feats_N_12[4] = extract_features(vectorizer_4_N2, train_text, validate_text)
feats_N_12[5] = extract_features(vectorizer_5_N2, train_text, validate_text)
print(feats_N_12[3][0].shape)
print(feats_N_12[4][0].shape)
print(feats_N_12[5][0].shape)
```

```
(5329, 4663)
(5329, 3353)
(5329, 2632)
```

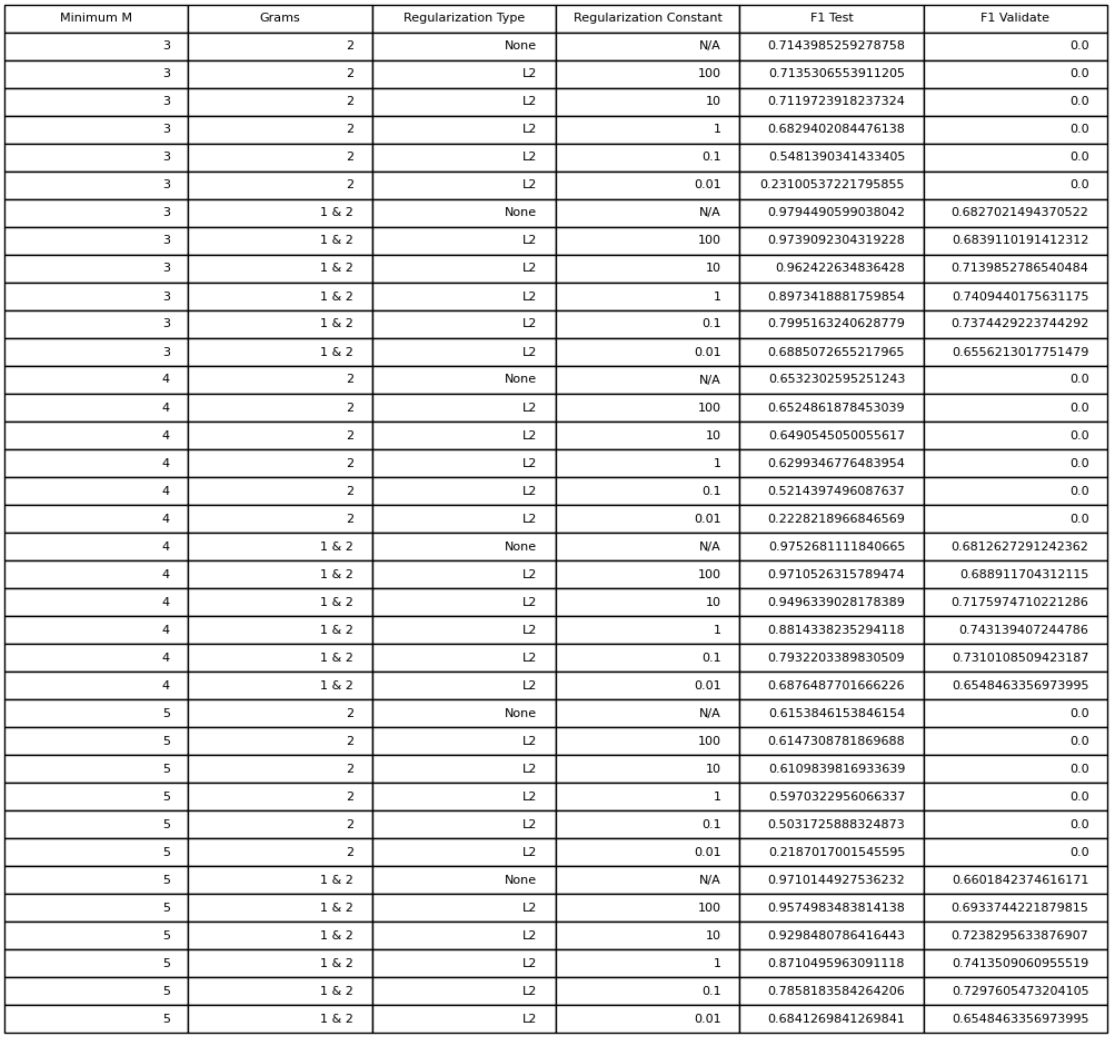
The results were surprising, because models trained only on 2-grams had an F1 score of 0 on validation sets while having decent F1 scores on training sets. This means that they had terrible generalization and overfitting was seen. I believe that this happened, because 2-grams like “about trap” frequencies are much rarer than just “about” or “trap”, which leads to each feature having one or few representatives. With poor representation on each feature, it is very easy to overfit. However, this alone does not explain the whole picture, since we observe F1 scores of 0 even when the regularization constant is extreme and training F1 scores drop. I believe that a feature set on 2-grams alone cannot work well on generalized samples, because the features created from 2-gram tokens are too specific.

For the models trained on both 1- and 2-grams, they behaved similar to the bag of words models. Similarly, overfitting was observed on weak or no regularization models and underfitting was observed on extreme regularization models. The best N-gram model was with the feature set using M=4, a combination of both 1 and 2-grams, and trained with an L2 regularization with constant of 1. The validation set F1 score is 0.743. This model is slightly worse than the best bag of words model. Thus, we see that the addition of 2-grams does not significantly help the classification of disaster tweets. This suggests that sequence-dependent words (at least for adjacent words) do not affect the semantic understanding of a tweet as disaster or not.

[230]:
```
# Run logistic regressions
data = [] # is (minimum M, N-gram, regularization type, reg constant, F1 test,␣
 ↪F1 validate)
reg_constants = [100,10,1,0.1,0.01]
for min_df in feats_N.keys():
    test, val = logistic_regression_noreg(feats_N[min_df][0],␣
 ↪feats_N[min_df][1])
    data.append((min_df, "2", "None", "N/A", test, val))
    for reg in reg_constants:
        test, val = logistic_regression_l2(feats_N[min_df][0],␣
 ↪feats_N[min_df][1], C=reg)
        data.append((min_df, "2", "L2", reg, test, val))
    test, val = logistic_regression_noreg(feats_N_12[min_df][0],␣
 ↪feats_N_12[min_df][1])
    data.append((min_df, "1 & 2", "None", "N/A", test, val))
    for reg in reg_constants:
        test, val = logistic_regression_l2(feats_N_12[min_df][0],␣
 ↪feats_N_12[min_df][1], C=reg)
        data.append((min_df, "1 & 2", "L2", reg, test, val))
```

```
Training F1: 0.7143985259278758, Validate F1: 0.0
Training F1: 0.7135306553911205, Validate F1: 0.0
Training F1: 0.7119723918237324, Validate F1: 0.0
Training F1: 0.6829402084476138, Validate F1: 0.0
Training F1: 0.5481390341433405, Validate F1: 0.0
Training F1: 0.23100537221795855, Validate F1: 0.0
Training F1: 0.9794490599038042, Validate F1: 0.6827021494370522
Training F1: 0.9739092304319228, Validate F1: 0.6839110191412312
Training F1: 0.962422634836428, Validate F1: 0.7139852786540484
Training F1: 0.8973418881759854, Validate F1: 0.7409440175631175
Training F1: 0.7995163240628779, Validate F1: 0.7374429223744292
Training F1: 0.6885072655217965, Validate F1: 0.6556213017751479
Training F1: 0.6532302595251243, Validate F1: 0.0
Training F1: 0.6524861878453039, Validate F1: 0.0
Training F1: 0.6490545050055617, Validate F1: 0.0
Training F1: 0.6299346776483954, Validate F1: 0.0
Training F1: 0.5214397496087637, Validate F1: 0.0
Training F1: 0.2228218966846569, Validate F1: 0.0
Training F1: 0.9752681111840665, Validate F1: 0.6812627291242362
Training F1: 0.9710526315789474, Validate F1: 0.688911704312115
Training F1: 0.9496339028178389, Validate F1: 0.7175974710221286
Training F1: 0.8814338235294118, Validate F1: 0.743139407244786
Training F1: 0.7932203389830509, Validate F1: 0.7310108509423187
Training F1: 0.6876487701666226, Validate F1: 0.6548463356973995
Training F1: 0.6153846153846154, Validate F1: 0.0
Training F1: 0.6147308781869688, Validate F1: 0.0
Training F1: 0.6109839816933639, Validate F1: 0.0
Training F1: 0.5970322956066337, Validate F1: 0.0
Training F1: 0.5031725888324873, Validate F1: 0.0
Training F1: 0.2187017001545595, Validate F1: 0.0
Training F1: 0.9710144927536232, Validate F1: 0.6601842374616171
Training F1: 0.9574983483814138, Validate F1: 0.6933744221879815
Training F1: 0.9298480786416443, Validate F1: 0.7238295633876907
Training F1: 0.8710495963091118, Validate F1: 0.7413509060955519
Training F1: 0.7858183584264206, Validate F1: 0.7297605473204105
Training F1: 0.6841269841269841, Validate F1: 0.6548463356973995
```

[231]:
```
# Graph all results
import numpy as np

arr = np.array(data)  # shape (n_rows, n_columns)
labels = ["Minimum M", "Grams", "Regularization Type", "Regularization␣
 ↪Constant", "F1 Test", "F1 Validate"]

fig, ax = plt.subplots()

ax.axis('off')  # hide axes

# Create table
table = ax.table(
    cellText=data,
    colLabels=labels,
    loc='center'
)

table.auto_set_font_size(True)
table.scale(2.0, 1.5)  # adjust cell sizes

plt.show()
```



Minimum M	Grams	Regularization Type	Regularization Constant	F1 Test	F1 Validate
3	2	None	N/A	0.7143985259278758	0.0
3	2	L2	100	0.7135306553911205	0.0
3	2	L2	10	0.7119723918237324	0.0
3	2	L2	1	0.6829402084476138	0.0
3	2	L2	0.1	0.5481390341433405	0.0
3	2	L2	0.01	0.23100537221795855	0.0
3	1 & 2	None	N/A	0.9794490599038042	0.6827021494370522
3	1 & 2	L2	100	0.9739092304319228	0.6839110191412312
3	1 & 2	L2	10	0.962422634836428	0.7139852786540484
3	1 & 2	L2	1	0.8973418881759854	0.7409440175631175
3	1 & 2	L2	0.1	0.7995163240628779	0.7374429223744292
3	1 & 2	L2	0.01	0.6885072655217965	0.6556213017751479
4	2	None	N/A	0.6532302595251243	0.0
4	2	L2	100	0.6524861878453039	0.0
4	2	L2	10	0.6490545050055617	0.0
4	2	L2	1	0.6299346776483954	0.0
4	2	L2	0.1	0.5214397496087637	0.0
4	2	L2	0.01	0.2228218966846569	0.0
4	1 & 2	None	N/A	0.9752681111840665	0.6812627291242362
4	1 & 2	L2	100	0.9710526315789474	0.688911704312115
4	1 & 2	L2	10	0.9496339028178389	0.7175974710221286
4	1 & 2	L2	1	0.8814338235294118	0.743139407244786
4	1 & 2	L2	0.1	0.7932203389830509	0.7310108509423187
4	1 & 2	L2	0.01	0.6876487701666226	0.6548463356973995
5	2	None	N/A	0.6153846153846154	0.0
5	2	L2	100	0.6147308781869688	0.0
5	2	L2	10	0.6109839816933639	0.0
5	2	L2	1	0.5970322956066337	0.0
5	2	L2	0.1	0.5031725888324873	0.0
5	2	L2	0.01	0.2187017001545595	0.0
5	1 & 2	None	N/A	0.9710144927536232	0.6601842374616171
5	1 & 2	L2	100	0.9574983483814138	0.6933744221879815
5	1 & 2	L2	10	0.9298480786416443	0.7238295633876907
5	1 & 2	L2	1	0.8710495963091118	0.7413509060955519
5	1 & 2	L2	0.1	0.7858183584264206	0.7297605473204105
5	1 & 2	L2	0.01	0.6841269841269841	0.6548463356973995

1.7 Determine performance with test set

The best model of overall was with the M=3 feature set using L2 regularization with C=1 with the bag of words model. The model was rebuilt using the entire training data. Recall that when this model was trained on the partial testing set, the F1 validation score was 0.75378.

```
[232]: # Get highest validation score model
max_model = max(data, key=lambda x: x[-1])
print(max_model)
```

```
(4, '1 & 2', 'L2', 1, 0.8814338235294118, 0.743139407244786)
```

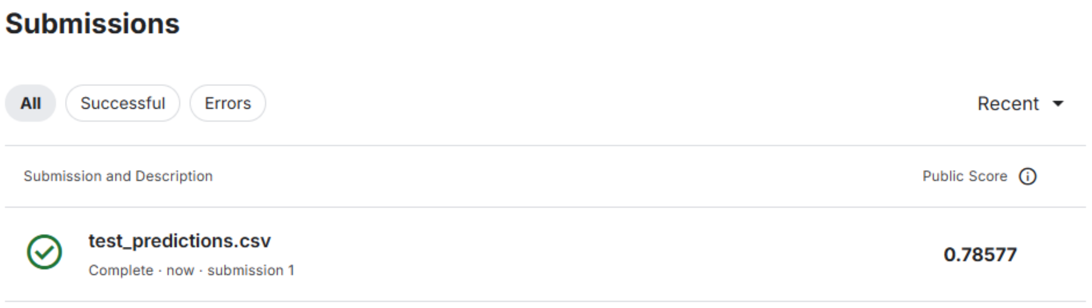
```
[241]: # Rebuild feature vectors and re-train on entire training set
# FINAL model choice: M=3, bag of words (no N-gram), L2, C=1
vectorizer_3 = CountVectorizer(binary=True, min_df=3)
train = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/train.csv")
```

```
validate = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/validate.
  ↪csv")
test = pd.read_csv("data-from-NLP-disaster-tweets/preprocessed/test.csv")

df_combined = pd.concat([train, validate], axis=0, ignore_index=True)

X = df_combined["text"]
Y = df_combined["target"]
X_test = test["text"]

X_feats, X_test_feats, words = extract_features(vectorizer_3, X, X_test)

clf = LogisticRegression(random_state=0, penalty="l2", max_iter=5000, C=1).
  ↪fit(X_feats, Y)
Y_pred = clf.predict(X_test_feats)
final_submission = pd.DataFrame({"id": test["id"], "target": Y_pred})
print(final_submission.head())
final_submission.to_csv("test_predictions.csv", index=False)
```

```
   id  target
0   0       1
1   2       1
2   3       1
3   9       1
4  11       1
```

When trained with the whole training set and submitted to Kaggle, the F1 score was 0.78577. So the F1 testing score was very similar to the validation score as expected. It was slightly higher. This very slight increase is likely because the validation set we used was only 2284 samples, but the testing set is 3263 samples. It is possible that our smaller validation set was a worse representation of the true distribution, but this slight variance could have been in either direction (so it is coincidence that the final testing score is higher). A more likely reason for the increase in F1 score is that the F1 validation score is based off predictions from the model trained from only 70% of the training data. However, the F1 testing score from Kaggle is based off predictions from the model trained from 100% of the training data. Greater number of sample helps the model determine the parameters closer to the true data generating model.

```
[ ]: # Submission F1 score was 0.78577
     # The same model trained on training set
     # and tested on validation set had F1 score of 0.7537796976241901
```



Submissions

Submission and Description	Public Score
✓ test_predictions.csv Complete · now · submission 1	0.78577

AML (CS5785) HW2

Jiwon Jeong

October 2025

1 Written Exercises

1.1 Maximum likelihood and KL divergence

Maximizing the likelihood that the model generates the data distribution and minimizing the KL divergence between the data distribution and the model predictions are the same. See Figure 12 for derivation.

1.2 Gradient and log-likelihood for logistic regression

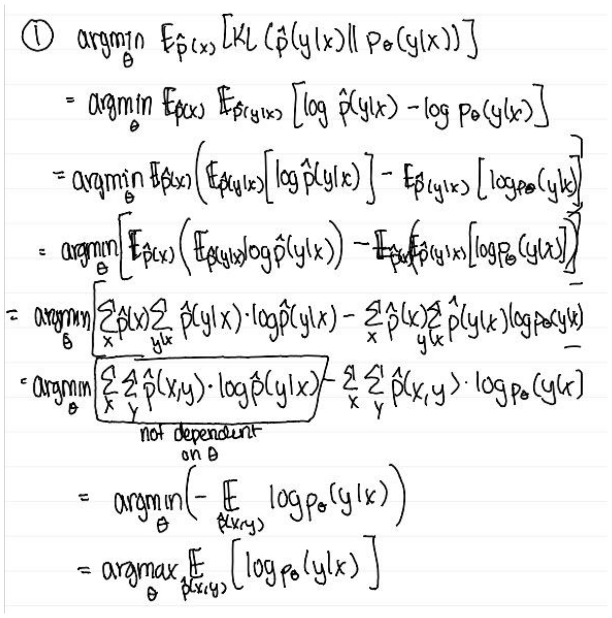
See Figure 13 and 14 for derivation on gradient of the log likelihood loss function.

1.3 Problem with single learning rate

Using a single learning rate for all components in a gradient descent means that all dimensions will use the same learning rate. This can be an issue because a too-high learning rate can break convergence, causing either divergence or jittering. Especially with higher dimensions, a single learning rate might mean that some dimensions converge well, but other dimensions cannot converge. One naive solution would be to just lower the single learning rate until all these dimensions converge, but that could mean that some dimensions now converge too slowly and will not reach convergence in a reasonable time.

1.4 Gradient descent can fail to reach global minimum

Previously alluded, gradient descent can fail to converge if the learning rate for that dimension is too high. There are two versions. If step size is extremely large, then the global minimum cannot be reached, because the steps will cause the solution to overstep and shoot out of convex loss surfaces. This is called divergence. Alternatively, the step size can still be large, causing the solution to step across the minimum to the other size of the convex surface and keep stepping around the minimum in the convex surface. While it may eventually converge, it will take a very long time. This is called jittering.



① $$\arg\min_\theta \mathbb{E}_{\hat{p}(x)}[KL(\hat{p}(y|x) \| p_\theta(y|x))]$$

$$= \arg\min_\theta \mathbb{E}_{\hat{p}(x)} \mathbb{E}_{\hat{p}(y|x)}[\log \hat{p}(y|x) - \log p_\theta(y|x)]$$

$$= \arg\min_\theta \mathbb{E}_{\hat{p}(x)}\left(\mathbb{E}_{\hat{p}(y|x)}[\log \hat{p}(y|x)] - \mathbb{E}_{\hat{p}(y|x)}[\log p_\theta(y|x)]\right)$$

$$= \arg\min_\theta \left[\mathbb{E}_{\hat{p}(x)}\left(\mathbb{E}_{\hat{p}(y|x)} \log \hat{p}(y|x)\right) - \mathbb{E}_{\hat{p}(x)}\left(\mathbb{E}_{\hat{p}(y|x)}[\log p_\theta(y|x)]\right)\right]$$

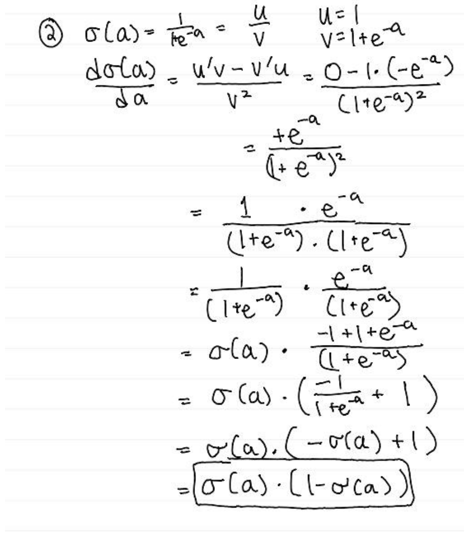
$$= \arg\min_\theta \left[\sum_x \hat{p}(x) \sum_{y|x} \hat{p}(y|x) \cdot \log \hat{p}(y|x) - \sum_x \hat{p}(x) \sum_{y|x} \hat{p}(y|x) \log p_\theta(y|x)\right]$$

$$= \arg\min_\theta \left[\underbrace{\boxed{\sum_x \sum_y \hat{p}(x,y) \cdot \log \hat{p}(y|x)}}_{\text{not dependent on } \theta} - \sum_x \sum_y \hat{p}(x,y) \cdot \log p_\theta(y|x)\right]$$

$$= \arg\min_\theta \left(- \mathbb{E}_{\hat{p}(x,y)} \log p_\theta(y|x)\right)$$

$$= \arg\max_\theta \mathbb{E}_{\hat{p}(x,y)}[\log p_\theta(y|x)]$$

Figure 1: Derivation for equality of minimizing KL divergence and maximum likelihood estimation



② $$\sigma(a) = \frac{1}{1+e^{-a}} = \frac{u}{v} \qquad u = 1 \qquad v = 1+e^{-a}$$

$$\frac{d\sigma(a)}{da} = \frac{u'v - v'u}{v^2} = \frac{0 - 1 \cdot (-e^{-a})}{(1+e^{-a})^2}$$

$$= \frac{+e^{-a}}{(1+e^{-a})^2}$$

$$= \frac{1 \cdot e^{-a}}{(1+e^{-a}) \cdot (1+e^{-a})}$$

$$= \frac{1}{(1+e^{-a})} \cdot \frac{e^{-a}}{(1+e^{-a})}$$

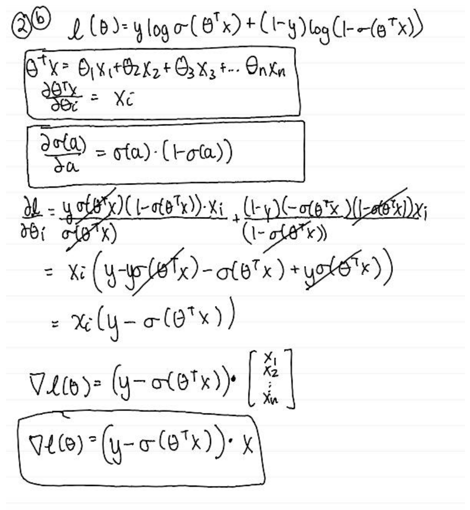
$$= \sigma(a) \cdot \frac{-1+1+e^{-a}}{(1+e^{-a})}$$

$$= \sigma(a) \cdot \left(\frac{-1}{1+e^{-a}} + 1\right)$$

$$= \sigma(a) \cdot (-\sigma(a) + 1)$$

$$= \boxed{\sigma(a) \cdot (1 - \sigma(a))}$$

Figure 2: Derivative of sigmoid function



2 b $\ell(\theta) = y \log \sigma(\theta^T x) + (1-y)\log(1-\sigma(\theta^T x))$

$$\boxed{\begin{aligned}\theta^T x &= \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots \theta_n x_n \\ \frac{\partial \theta^T x}{\partial \theta_i} &= x_i\end{aligned}}$$

$$\boxed{\frac{\partial \sigma(a)}{\partial a} = \sigma(a)\cdot(1-\sigma(a))}$$

$$\frac{\partial \ell}{\partial \theta_i} = \frac{y\,\cancel{\sigma(\theta^T x)}(1-\sigma(\theta^T x))\cdot x_i}{\cancel{\sigma(\theta^T x)}} + \frac{(1-y)(-\sigma(\theta^T x))\cancel{(1-\sigma(\theta^T x))}x_i}{\cancel{(1-\sigma(\theta^T x))}}$$

$$= x_i\left(y - \cancel{y\sigma(\theta^T x)} - \sigma(\theta^T x) + \cancel{y\sigma(\theta^T x)}\right)$$

$$= x_i(y - \sigma(\theta^T x))$$

$$\nabla \ell(\theta) = (y - \sigma(\theta^T x))\cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$\boxed{\nabla \ell(\theta) = (y - \sigma(\theta^T x))\cdot x}$$

Figure 3: Gradient of log likelihood

However, gradient descent can also fail to reach global minima under other conditions. If the learning rate is too low and/or the loss surface is very flat in regions away from the minimum, then the solution will step so slowly that the minimum is never reached in realistic time. Particularly concerning this idea of the loss surface being very flat, the loss surface can have flat points or stationary points if there are many collinear features.

Thus, ideally the step size is optimized to not be too large for divergence/jittering or too small for unrealistic convergence times. In fact for convex loss surfaces, there is an optimal step size for which we can reach the minimum in one step.

However, even gradient descents with optimized step sizes can still fail to reach a global minimum. This can happen for two main reasons. First, if there are local minima, it is possible that the specific initialization brings the solution to converge to a local minimum. This is the most concern for nonconvex loss surfaces. Second, if parts of the loss surface is nondifferentiable, the vanilla computation for gradient descent can get stuck at nondifferentiable points.

1.5 Decaying learning rate

When getting close to a (hopefully global) minimum and within the local convex surface, we need a small step size to guarantee convergence. Otherwise, a large step size could cause divergence or very slow convergence through jittering.

However, having a small step size throughout the whole optimization can be a bad idea, because we want to quickly approach the minima. The loss surface might be relatively flat when farther away from the minima, so a larger step size is necessary to reach the minima quickly. In addition, for complex non-convex loss surfaces, a large initial step size can help escape local minima.

Thus, we initially set a high learning rate to take faster steps towards the minima and possibly escape local minima. Then we gradually decrease the learning rate, so that it is eventually possible to converge in a local convex surface towards the (hopefully global) minimum. This change in learning rate is typically performed through 3 decay schedules. Starting from a large learning rate of greater than 2, the learning rate is decreased with each training loop iteration. The linear decay schedule decreases rate based on number of iterations in a $1/x$ rational decay. The quadratic decay schedule decreases rate based on number of iterations in a $1/x^2$ rational decay. The exponential decay schedule decreases rate based on number of iterations in a $1/e^x$ exponential decay.

Using these decaying learning rates means that convergence will initially not occur. There will be divergence away from (local) minima and large step sizes. With iterations, the steps will get smaller and hopefully allow the solution to approach a minima. It is important to choose the right decay schedule so that the step size does not decay too quickly (leading to slow optimization from too small steps) or decay too slowly (leading to slow optimization from unnecessary divergence/jittering).