SVM implementation

```
import numpy as np
from copy import copy
import pandas as pd
import math
import random
import matplotlib.pyplot as plt
from tqdm import tqdm
%matplotlib inline
```

Initializing random seed and global path variables

```
# Initializing

np.random.seed(42)
TRAINING_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-deci
TESTING_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decis
EVAL_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decision

TRAINING_PATH_GLOVE = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-baile
TESTING_PATH_GLOVE = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey
EVAL_PATH_GLOVE = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-de

TRAINING_PATH_TF = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-d
TESTING_PATH_TF = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-de
EVAL_PATH_TF = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decis

TRAINING_PATH_BOW = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-
TESTING_PATH_BOW = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-d
EVAL_PATH_BOW = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-deci

EVAL_IDS = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions

# Feature vector paths
X_TRAIN = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/
y_TRAIN = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/

X_TEST = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/e
y_TEST = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/e

X_EVAL = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/e
y_EVAL = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/e
```

Data class

w/ some other helper functions

```
# Defining Data class
# Will define new data class using csv and numpy

class Data:
  def __init__(self,file_path=None):
    if file_path != None:
      self.raw_data,\
      self.y,\
      self.X,\
      self.num_examples,\
      self.num_features = self.load_data_from_path(file_path)

  def load_data_from_path(self,file_path):
    # data = np.loadtxt(file_path, delimiter = ",")
    raw_data = pd.read_csv(file_path)
    data = raw_data.to_numpy()
    labels = data[:,0]
    instances = data[:,1:]

    # Add a 1 to the end of each instance
    bias = np.ones((data.shape[0],1))
    instances = np.append(instances,bias,axis=1)

    num_examples = data.shape[0]
    num_features = instances.shape[1]
    return data,labels,instances,num_examples,num_features

  def load_data(self,raw_data):
    self.raw_data = raw_data
    self.y = raw_data[:,0]
    self.y[self.y == 0] = - 1
    instances = raw_data[:,1:]
    # Add a 1 to the end of each instance
    bias = np.ones((raw_data.shape[0],1))
    self.X = np.append(instances,bias,axis=1)
    self.num_examples = raw_data.shape[0]
    self.num_features = self.X.shape[1]

  # For loading in feature vectors from NN
  def load_data_np(self,X,y):
    y[y==0] = -1
    self.raw_data = np.append(y,X,axis=1)
    bias = np.ones((X.shape[0],1))
    self.X = np.append(X,bias,axis=1)
    self.y = np.ravel(y)
    self.num_examples = X.shape[0]
    self.num_features = self.X.shape[1]

  def add_bias_to_features(self):
    # Add a 1 to the end of each instance
    bias = np.ones((self.num_examples,1))
    self.X = np.append(self.X,bias,axis=1)

  def add_data(self,data):
```

```
  def add_data(self,data):
    # takes as input another data object and adds that data to this object
    self.raw_data = np.vstack((self.raw_data,data.raw_data))
    self.X = np.vstack((self.X,data.X))
    self.y = np.hstack((self.y,data.y))
    self.num_examples += data.num_examples

  # returns shuffled labels and instances
  def shuffle_data(self):
    shuffled_raw_data = np.copy(self.raw_data)
    np.random.shuffle(shuffled_raw_data)
    shuffled_labels = shuffled_raw_data[:,0]
    shuffled_instances = shuffled_raw_data[:,1:]
    # add in bias
    bias = np.ones((shuffled_raw_data.shape[0],1))
    shuffled_instances = np.append(shuffled_instances,bias,axis=1)

    return shuffled_instances,shuffled_labels


# plot learning curve
def plot_learning(x,y,title,x_label,y_label):
  # Let's plot
  plt.style.use('default')
  plt.rcParams['font.family'] = 'Avenir'
  plt.figure(figsize = (11,4.5))
  # My PCA
  plt.plot(x,y)
  plt.title(title,fontsize=15)
  plt.xlabel(x_label)
  plt.ylabel(y_label)
  [i.set_linewidth(0.4) for i in plt.gca().spines.values()]
```

SVM class from hw6, may need to make some adjustments to accomodate this dataset.

```
# SVM Class

class SVM:
  def __init__(self):
    self.W = None
    self.Weights = {} # init empty dict of Weights, add to this for each epoch
    self.accuracies = {} # init empty dict of accuracies, which I store at end of each epoch
    self.loss = {} # dictionary contatining loss at each step
    self.num_updates = 0 # records number of updates made

  def initialize_weights(self,num_features):
    self.W = np.array([np.random.uniform(-0.01,0.01) for _ in range(num_features)])
    # self.W = np.zeros((num_features)) # init to zeros

  def train(self,data,epochs=1,learning_rate=1,reg_strength=1):
    C = reg_strength
    epochs = epochs
    N = data.num_examples
    D = data.num_features
```

```
    D = data.num_features
    # print("N:",N,"D (including b):",D)
    # initialize weights
    self.initialize_weights(D)

    for t in range(epochs):
      lr = learning_rate / (1 + t) # we use a decaying learning rate
      # shuffle data
      X,y = data.shuffle_data()
      # loop over each example in the training set
      for i in range(N):
        v = y[i]*(self.W.T.dot(X[i]))
        # print(v)
        if v <= 1.0:
          self.W = (1.0-lr)*self.W + (lr*C*y[i])*X[i]
        else:
          self.W = (1.0-lr)*self.W
      # store this iteration of weights
      self.Weights[t] = self.W
      # store the accuracy of these weights
      self.accuracies[t] = self.get_accuracy_own_weights(data,self.W)
      # Compute and store the loss
      self.loss[t] = self.compute_loss(data,self.W,C)


  # Helper methods for predicting and accuracy
  def get_best_weights_and_bias(self):
    # print(self.accuracies.items())
    best_epoch = max(self.accuracies,key=self.accuracies.get)
    # print("best epoch: ",best_epoch)
    return self.Weights[best_epoch],best_epoch

  def predict(self,data):
    predictions = np.sign(data.dot(self.W))
    return predictions

  def get_predict_accuracy(self,data):
    predictions = self.predict(data.X)
    equal = np.equal(predictions,data.y)
    return np.sum(equal)/data.num_examples

  def get_accuracy_own_weights(self,data,W):
    predictions = np.sign(data.X.dot(W)) # Should the prediction have a margin? No, I don't th
    equal = np.equal(predictions,data.y)
    return np.sum(equal)/data.num_examples

  def compute_loss(self,data,W,C):
    # "Loss" of the entire dataset
    X = data.X
    y = data.y
    loss = 0.5*(W.T.dot(W))
    a = 1 - y*W.dot(X.T)
    a[a<0] = 0
    return loss + C*np.sum(a)
```

Load data

Takes a while b/c I'm loading from csv.

```
# TF-IDF + Misc
# train_data = Data(TRAINING_PATH_TF)
# test_data = Data(TESTING_PATH_TF)

# NN feature vecs
train_data = Data()
test_data = Data()
eval_data = Data()

X_train = np.load(X_TRAIN)
y_train = np.load(y_TRAIN)
X_test = np.load(X_TEST)
y_test = np.load(y_TEST)
X_eval = np.load(X_EVAL)
y_eval = np.load(y_EVAL)

train_data.load_data_np(X_train,y_train)
test_data.load_data_np(X_test,y_test)
eval_data.load_data_np(X_eval,y_eval)


# Glove + Misc
# train_data = Data(TRAINING_PATH_GLOVE)
# test_data = Data(TESTING_PATH_GLOVE)

# BOW + Misc
# train_data = Data(TRAINING_PATH_BOW)
# test_data = Data(TESTING_PATH_BOW)

# # Misc
# misc_train_data = Data(TRAINING_PATH)
# misc_test_data = Data(TESTING_PATH)
```

```
train_data.y
```

```
array([ 1.,  1., -1., ...,  1.,  1.,  1.])
```

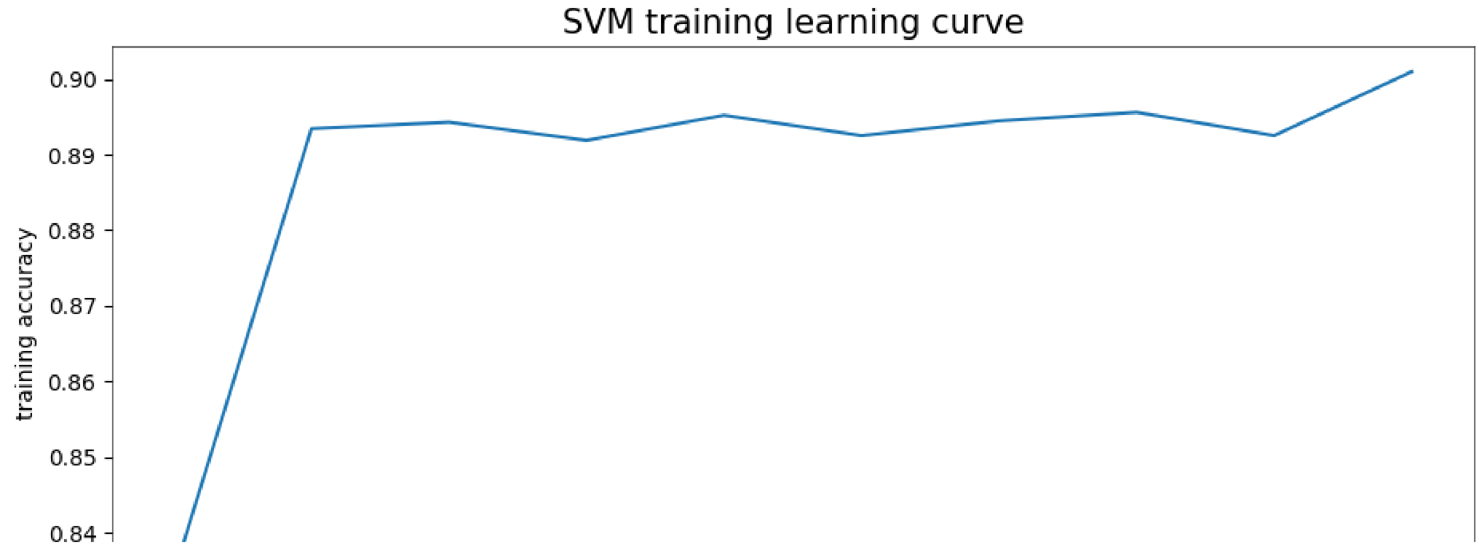
Initial training

Try some params I think may be good and see how long it takes/ how good the params are.

```
learning_rate = 0.00001
C = 10000
```

```
epochs = 10
svm = SVM()
%time svm.train(train_data,epochs,learning_rate,C)

# test set accuracy
# Get the best weights and bias from this training
W,best_epoch = svm.get_best_weights_and_bias()
# training set accuracy:
print("best training set accuracy: ", svm.accuracies[best_epoch] )
# Use these weights and bias to evaluate on the test set
test_accuracy = svm.get_accuracy_own_weights(test_data,W)
print("final test accuracy: ",test_accuracy)

y = list(svm.accuracies.values())
x = [i for i in range(epochs)]
title = 'SVM training learning curve'
plot_learning(x,y,title,'epochs','training accuracy')

y = list(svm.loss.values())
x = [i for i in range(epochs)]
title = 'SVM training loss'
plot_learning(x,y,title,'epochs','loss')
```

```
CPU times: user 1.63 s, sys: 894 ms, total: 2.52 s
Wall time: 1.45 s
best training set accuracy:  0.9009714285714285
final test accuracy:  0.8404444444444444
```



Make cross validation folds

```
# Validation splits - split training data into k splits

k = 3
folds = np.array_split(train_data.raw_data,k)
```

Cross validate

Will use tf-idf + misc attributes b/c this is what's worked the best in the past. Once I get the best params, will run on other datasets just to confirm.

Not using an exhaustive search b/c this takes FOREVER.

```
def cross_validate(epochs,folds,learning_rates,regularizations,verbose=False,model=SVM()):

  # dictionaries storing accuracies corresponding to certain hyper parameter combinations
  mean_accuracies = {}
  standard_deviations = {}

  num_combs = len(learning_rates)*len(regularizations)
  progress = 0

  for lr in tqdm(learning_rates):
    for C in tqdm(regularizations):
      accuracies = []
      # Need to concatenate 4 of the folds into one training set and leave out one as my test
      for i in tqdm(range(k)):
        # Initialize new data objects
        val_data = Data()
        train_data = Data()
        folds_copy = list.copy(folds)
```

```
        # Set validation data
        val_data.load_data(np.array(folds_copy.pop(i)))
        # set training data
        train_data.load_data(np.concatenate(folds_copy,axis=0))
        # train on folds
        svm = model
        svm.train(train_data,epochs,lr,C)
        weights,best_epoch = svm.get_best_weights_and_bias()
        # calculate validation accuracy
        val_accuracy = svm.get_accuracy_own_weights(val_data,weights)
        accuracies.append(val_accuracy)

      mean_accuracies[(lr,C)] = np.mean(accuracies)
      standard_deviations[(lr,C)] = np.std(accuracies)
      if verbose == True:
        print("accuracy: ",mean_accuracies[(lr,C)],"lr: ",lr,"C: ",C)
        # print("list",accuracies)
        progress += 1
        print("{:.4}% complete".format(100*progress/num_combs))

  print(mean_accuracies.items())
  print(standard_deviations.items())
  best_vals = max(mean_accuracies,key=mean_accuracies.get)
  print("best lr: ",best_vals[0],"best C: ",best_vals[1],"cross-val accuracy: ",mean_accuracie
  return best_vals
```

```
# cross validate svm

epochs = 4
# learning_rates = [10**0, 10**-1, 10**-2, 10**-3, 10**-4]
learning_rates = [10**0, 10**-2, 10**-4, 10**-4]
# regularizations = [10**3, 10**2, 10**1, 10**0, 10**-1, 10**-2]
regularizations = [10**5, 10**4, 10**3, 10**-1, 10**-2]

best_vals = cross_validate(epochs,folds,learning_rates,regularizations,verbose=True)
```

```
      0%|          | 0/4 [00:00<?, ?it/s]
      0%|          | 0/5 [00:00<?, ?it/s]

      0%|          | 0/3 [00:00<?, ?it/s]

     33%|███       | 1/3 [00:00<00:00,  2.23it/s]

     67%|██████    | 2/3 [00:00<00:00,  2.19it/s]

    100%|██████████| 3/3 [00:01<00:00,  2.11it/s]

     20%|██        | 1/5 [00:01<00:05,  1.43s/it]

      0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.8877715780340493 lr:  1 C:  100000
    5.0% complete


     33%|███       | 1/3 [00:00<00:00,  2.17it/s]

     67%|██████    | 2/3 [00:00<00:00,  2.14it/s]
```

```
100%|██████████| 3/3 [00:01<00:00,  2.12it/s]

 40%|████      | 2/5 [00:02<00:04,  1.43s/it]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.8872002441473565 lr:  1 C:  10000
10.0% complete


 33%|███▎      | 1/3 [00:00<00:00,  2.13it/s]

 67%|██████▋   | 2/3 [00:00<00:00,  2.13it/s]

100%|██████████| 3/3 [00:01<00:00,  2.07it/s]

 60%|██████    | 3/5 [00:04<00:02,  1.44s/it]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.887714451502131 lr:  1 C:  1000
15.0% complete


 33%|███▎      | 1/3 [00:00<00:00,  2.00it/s]

 67%|██████▋   | 2/3 [00:01<00:00,  1.98it/s]

100%|██████████| 3/3 [00:01<00:00,  1.98it/s]

 80%|████████  | 4/5 [00:05<00:01,  1.46s/it]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.8874287110935944 lr:  1 C:  0.1
20.0% complete


 33%|███▎      | 1/3 [00:00<00:01,  1.83it/s]

 67%|██████▋   | 2/3 [00:01<00:00,  1.84it/s]

100%|██████████| 3/3 [00:01<00:00,  1.82it/s]
```

TF-IDF + misc data

From this limited cross val I got:

best lr: 0.0001 best C: 1000 cross-val accuracy: 0.790914307204743

However, it's worth noting that for every other learning rate, lower C's worked better. I'm going to hand test a few other values exploring this.

lr: 0.0001; C = 0.01: acc: 0.782514285714285.
lr: 0.0001; C = 10000: acc: 0.8330285714285715.
lr: 0.0001; C = 100000 acc: 0.84.
lr: 0.00001; C = 100000; accc: 0.89.

Seems that smaller learning rates and bigger C's do pretty well. Let's try cross val with a slightly different range.

Feat Vec Data

best lr: 0.0001 best C: 1000 cross-val accuracy: 0.8947429837951658

```
# cross validate svm

epochs = 4
# learning_rates = [10**0, 10**-1, 10**-2, 10**-3, 10**-4]
learning_rates = [10**-4, 10**-5, 10**-6]
# regularizations = [10**3, 10**2, 10**1, 10**0, 10**-1, 10**-2]
regularizations = [10**5, 10**4, 10**3]

best_vals = cross_validate(epochs,folds,learning_rates,regularizations,verbose=True)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
  0%|          | 0/3 [00:00<?, ?it/s]

  0%|          | 0/3 [00:00<?, ?it/s]

 33%|███▎      | 1/3 [00:47<01:34, 47.33s/it]

 67%|██████▋   | 2/3 [01:32<00:46, 46.70s/it]

100%|██████████| 3/3 [02:17<00:00, 45.71s/it]

 33%|███▎      | 1/3 [02:17<04:34, 137.14s/it]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.7905702060490923 lr:  0.0001 C:  1000
11.11% complete


 33%|███▎      | 1/3 [00:46<01:33, 46.94s/it]

 67%|██████▋   | 2/3 [01:32<00:46, 46.56s/it]

100%|██████████| 3/3 [02:18<00:00, 46.04s/it]

 67%|██████▋   | 2/3 [04:35<02:17, 137.43s/it]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.8006281881076392 lr:  0.0001 C:  1000
22.22% complete


 33%|███▎      | 1/3 [00:48<01:36, 48.11s/it]

 67%|██████▋   | 2/3 [01:34<00:47, 47.67s/it]

100%|██████████| 3/3 [02:21<00:00, 47.06s/it]

100%|██████████| 3/3 [06:56<00:00, 138.82s/it]
 33%|███▎      | 1/3 [06:56<13:52, 416.48s/it]
  0%|          | 0/3 [00:00<?, ?it/s]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.7946287072224686 lr:  0.0001 C:  1000
33.33% complete


 33%|███▎      | 1/3 [00:46<01:32, 46.30s/it]

 67%|██████▋   | 2/3 [01:31<00:45, 45.91s/it]
```

```
100%|██████████| 3/3 [02:16<00:00, 45.53s/it]

 33%|███       | 1/3 [02:16<04:33, 136.60s/it]

  0%|          | 0/3 [00:00<?, ?it/s]accuracy:  0.8166283971797753 lr:  1e-05 C:  10000
44.44% complete


 33%|███       | 1/3 [00:47<01:35, 47.84s/it]

 67%|██████▋   | 2/3 [01:33<00:47, 47.23s/it]
```

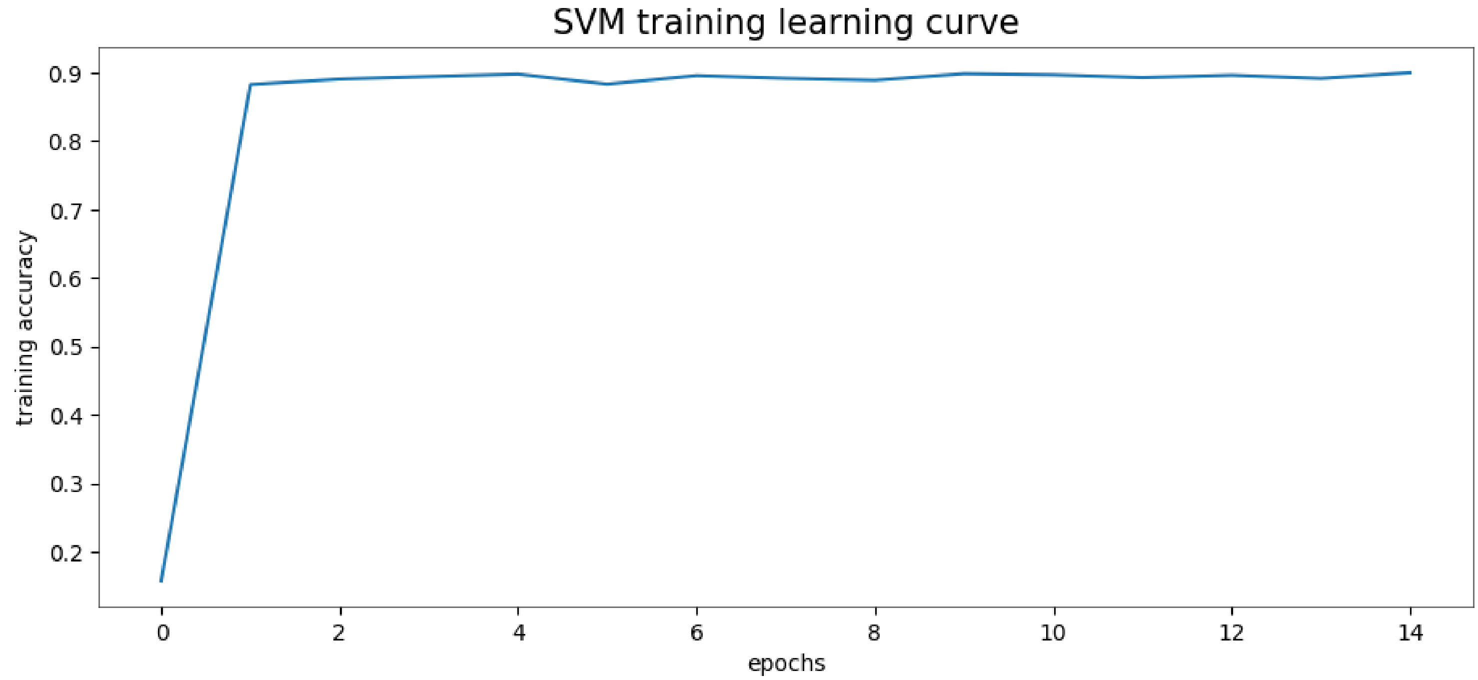
Ok, this second cross validation yielded:

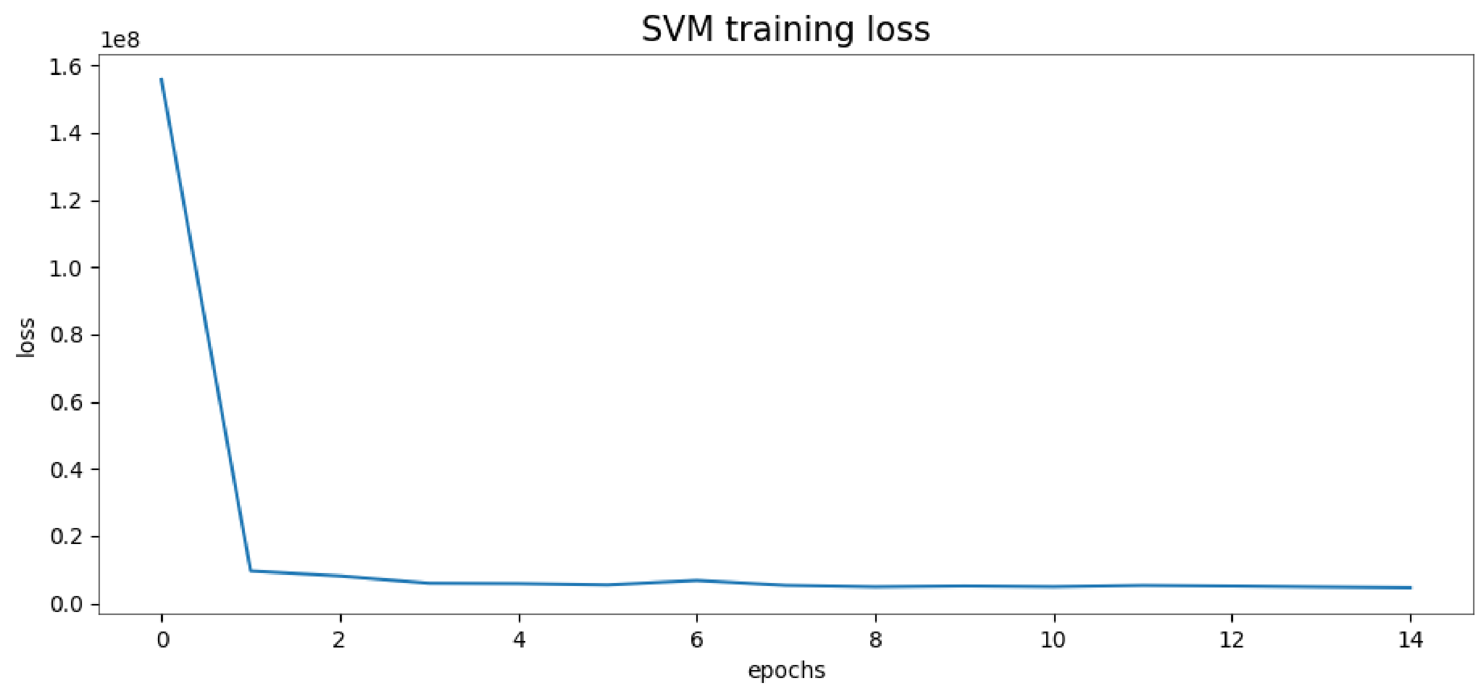
best lr: 1e-05 best C: 100000 cross-val accuracy: 0.8166283971797753.

So, let's train with these params.

```
# TF-IDF + misc data
# learning_rate = 0.00001
# C = 100000

# Feat vec data
learning_rate = 0.0001
C = 1000
#best lr:  0.0001 best C:  1000 cross-val accuracy:  0.8947429837951658

epochs = 15
svm = SVM()
%time svm.train(train_data,epochs,learning_rate,C)

# test set accuracy
# Get the best weights and bias from this training
W,best_epoch = svm.get_best_weights_and_bias()
# training set accuracy:
print("best training set accuracy: ", svm.accuracies[best_epoch] )
# Use these weights and bias to evaluate on the test set
test_accuracy = svm.get_accuracy_own_weights(test_data,W)
print("final test accuracy: ",test_accuracy)

y = list(svm.accuracies.values())
x = [i for i in range(epochs)]
title = 'SVM training learning curve'
plot_learning(x,y,title,'epochs','training accuracy')

y = list(svm.loss.values())
x = [i for i in range(epochs)]
title = 'SVM training loss'
plot_learning(x,y,title,'epochs','loss')
```

```
CPU times: user 2.47 s, sys: 1.41 s, total: 3.88 s
Wall time: 2.19 s
best training set accuracy:  0.9001714285714286
final test accuracy:  0.8342222222222222
```



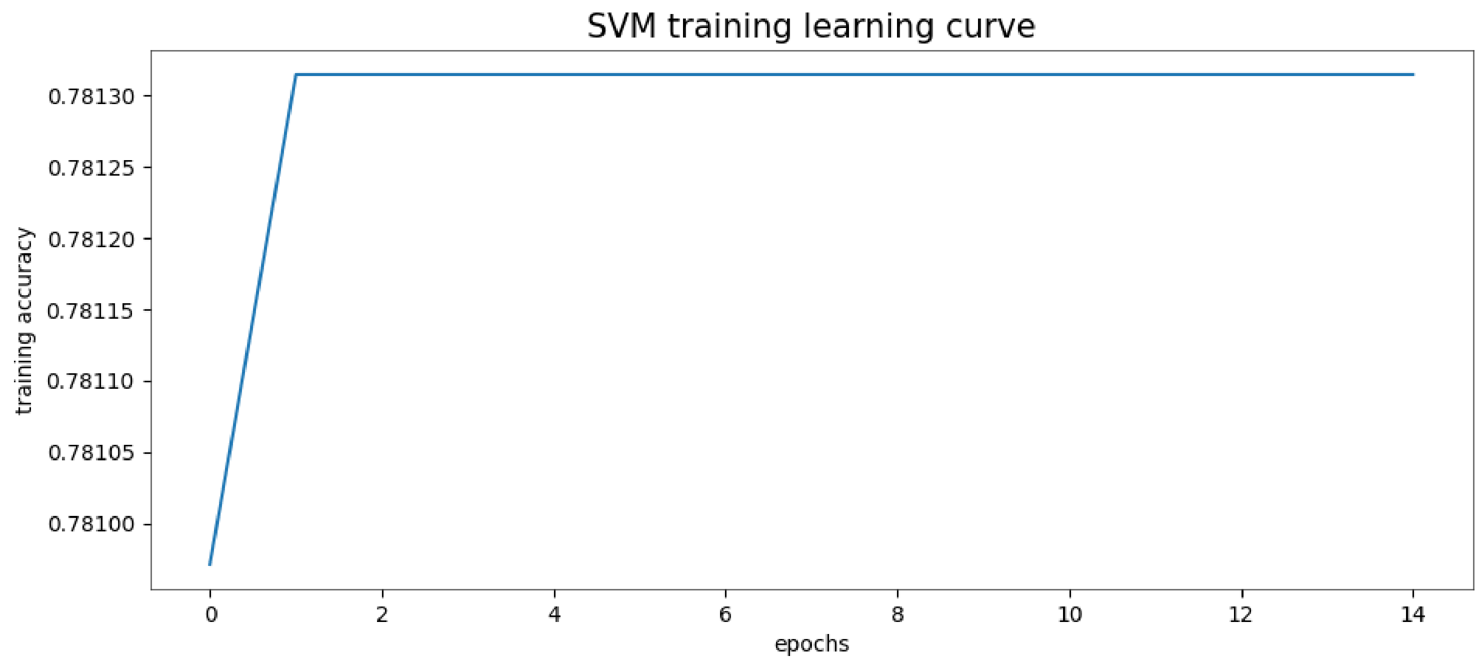


I wonder if using any of the other datasets would perform better? Let's try a few.

```
# Misc
misc_train_data = Data(TRAINING_PATH)
misc_test_data = Data(TESTING_PATH)
```

```
learning_rate = 0.001
C = 100

epochs = 15
svm = SVM()
%time svm.train(misc_train_data,epochs,learning_rate,C)

# test set accuracy
# Get the best weights and bias from this training
W_misc,best_epoch_misc = svm.get_best_weights_and_bias()
# training set accuracy:
print("best training set accuracy: ", svm.accuracies[best_epoch_misc] )
# Use these weights and bias to evaluate on the test set
test_accuracy = svm.get_accuracy_own_weights(misc_test_data,W_misc)
print("final test accuracy: ",test_accuracy)

y = list(svm.accuracies.values())
x = [i for i in range(epochs)]
title = 'SVM training learning curve'
plot_learning(x,y,title,'epochs','training accuracy')

y = list(svm.loss.values())
x = [i for i in range(epochs)]
title = 'SVM training loss'
plot_learning(x,y,title,'epochs','loss')
```

```
CPU times: user 2.71 s, sys: 1.4 s, total: 4.11 s
Wall time: 2.42 s
best training set accuracy:  0.7813142857142857
final test accuracy:  0.7902222222222223
```



Misc data actually looks kind of good... let's run some cross val on it?

```
# Validation splits - split training data into k splits

k = 4
folds = np.array_split(misc_train_data.raw_data,k)

# cross validate svm

epochs = 10
# learning_rates = [10**0, 10**-1, 10**-2, 10**-3, 10**-4]
learning_rates = [10**1, 10**0, 10**-2, 10**-4, 10**-5]
# regularizations = [10**3, 10**2, 10**1, 10**0, 10**-1, 10**-2]
regularizations = [10**5, 10**4, 10**3, 10**1, 10**-1, 10**-2]

best_vals_misc = cross_validate(epochs,folds,learning_rates,regularizations,verbose=True)
```

```
  0%|          | 0/5 [00:00<?, ?it/s]
  0%|          | 0/6 [00:00<?, ?it/s]

  0%|          | 0/4 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-packages/ipykernel_l
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:71: RuntimeWarning: invali
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:35: RuntimeWarning: invali


 25%|██        | 1/4 [00:01<00:03,  1.17s/it]

 50%|████      | 2/4 [00:02<00:02,  1.17s/it]/usr/local/lib/python3.6/dist-packages/ipy


 75%|██████    | 3/4 [00:03<00:01,  1.18s/it]

100%|████████| 4/4 [00:04<00:00,  1.19s/it]
```

```
 17%|█         | 1/6 [00:04<00:23,  4.75s/it]

  0%|          | 0/4 [00:00<?, ?it/s]accuracy:  0.0 lr:  10 C:  100000
3.333% complete


 25%|██▌       | 1/4 [00:01<00:03,  1.20s/it]

 50%|█████     | 2/4 [00:02<00:02,  1.19s/it]

 75%|███████▌  | 3/4 [00:03<00:01,  1.19s/it]

100%|██████████| 4/4 [00:04<00:00,  1.19s/it]

 33%|███▎      | 2/6 [00:09<00:19,  4.77s/it]

  0%|          | 0/4 [00:00<?, ?it/s]accuracy:  0.0 lr:  10 C:  10000
6.667% complete


 25%|██▌       | 1/4 [00:01<00:03,  1.20s/it]

 50%|█████     | 2/4 [00:02<00:02,  1.20s/it]

 75%|███████▌  | 3/4 [00:03<00:01,  1.19s/it]

100%|██████████| 4/4 [00:04<00:00,  1.19s/it]

 50%|█████     | 3/6 [00:14<00:14,  4.77s/it]

  0%|          | 0/4 [00:00<?, ?it/s]accuracy:  0.0 lr:  10 C:  1000
10.0% complete


 25%|██▌       | 1/4 [00:01<00:03,  1.20s/it]

 50%|█████     | 2/4 [00:02<00:02,  1.20s/it]

 75%|███████▌  | 3/4 [00:03<00:01,  1.20s/it]

100%|██████████| 4/4 [00:04<00:00,  1.20s/it]
```

w/ this cross validation I get:

best lr: 1e-05 best C: 10000 cross-val accuracy: 0.7888.

So, train on these params

```
learning_rate = 0.00001
C = 100000

epochs = 15
svm = SVM()
%time svm.train(misc_train_data,epochs,learning_rate,C)

# test set accuracy
# Get the best weights and bias from this training
W_misc,best_epoch_misc = svm.get_best_weights_and_bias()
```

```
# training set accuracy:
print("best training set accuracy: ", svm.accuracies[best_epoch_misc] )
# Use these weights and bias to evaluate on the test set
test_accuracy = svm.get_accuracy_own_weights(misc_test_data,W_misc)
print("final test accuracy: ",test_accuracy)

y = list(svm.accuracies.values())
x = [i for i in range(epochs)]
title = 'SVM training learning curve'
plot_learning(x,y,title,'epochs','training accuracy')

y = list(svm.loss.values())
x = [i for i in range(epochs)]
title = 'SVM training loss'
plot_learning(x,y,title,'epochs','loss')
```

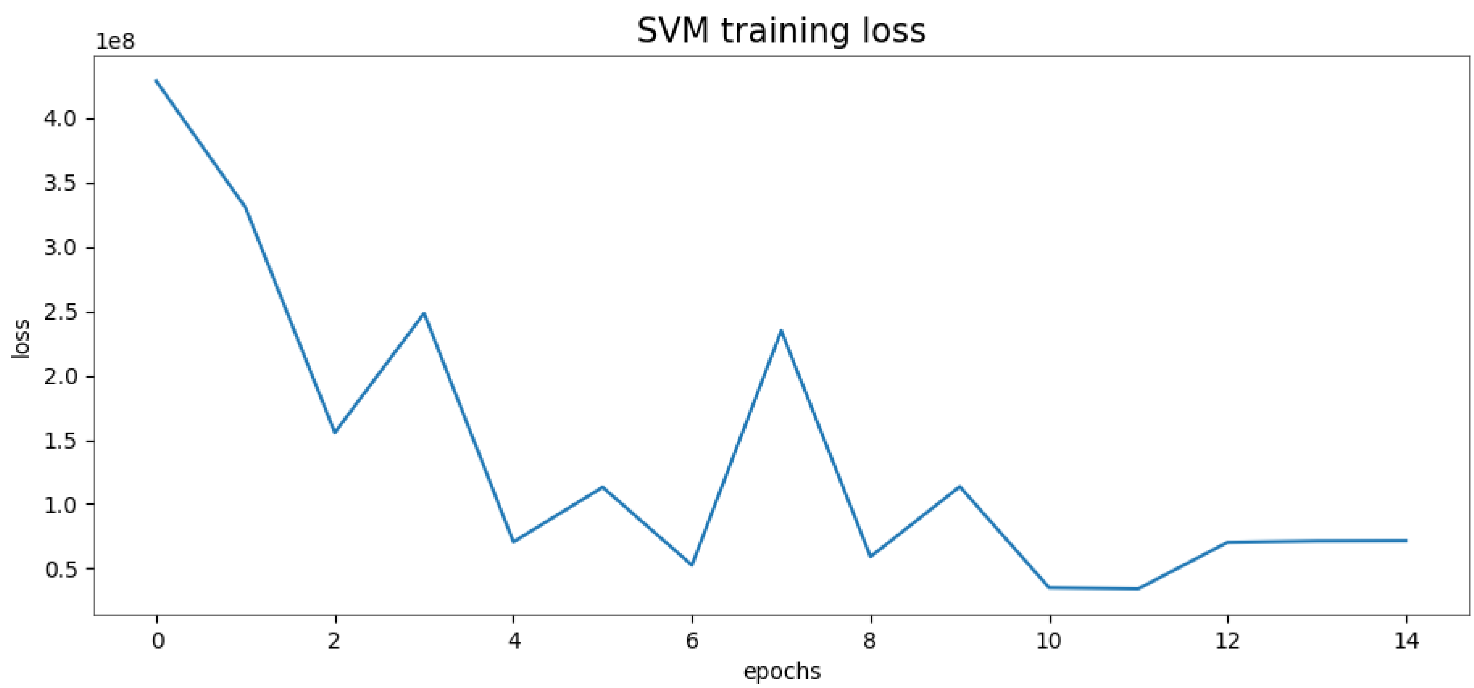
```
CPU times: user 2.73 s, sys: 1.36 s, total: 4.09 s
Wall time: 2.44 s
best training set accuracy:  0.7909714285714285
final test accuracy:  0.8
```

Eh, not great, I'll try glove next.

```
# Glove + Misc
train_data_glove = Data(TRAINING_PATH_GLOVE)
test_data_glove = Data(TESTING_PATH_GLOVE)
```

```
learning_rate = 0.001
C = 1000

epochs = 15
svm = SVM()
%time svm.train(train_data_glove,epochs,learning_rate,C)

# test set accuracy
# Get the best weights and bias from this training
W_glove,best_epoch_misc = svm.get_best_weights_and_bias()
# training set accuracy:
print("best training set accuracy: ", svm.accuracies[best_epoch_misc] )
# Use these weights and bias to evaluate on the test set
test_accuracy = svm.get_accuracy_own_weights(test_data_glove,W_glove)
print("final test accuracy: ",test_accuracy)

y = list(svm.accuracies.values())
x = [i for i in range(epochs)]
title = 'SVM training learning curve'
plot_learning(x,y,title,'epochs','training accuracy')

y = list(svm.loss.values())
x = [i for i in range(epochs)]
title = 'SVM training loss'
plot_learning(x,y,title,'epochs','loss')
```

```
CPU times: user 4.66 s, sys: 1.39 s, total: 6.05 s
Wall time: 4.28 s
best training set accuracy:  0.7778857142857143
final test accuracy:  0.7688888888888888
```



eh, not that great either. Will just stick with the tf-idf + misc features.

Run model on eval set

```
# load eval data
eval_data = Data(EVAL_PATH_TF)
```

```
# eval ids
def load_ids(file_path):
  with open(file_path) as f:
    raw_data = [int(line.split()[0]) for line in f]
```

```
    raw_data = [int(line.split()[0]) for line in f]
  # print(raw_data)
  return raw_data

eval_ids = np.reshape(np.array(load_ids(EVAL_IDS),dtype=np.int32),(eval_data.X.shape[0],1))
# print(eval_ids)

# run predictions
predictions = np.sign(eval_data.X.dot(W))
print(predictions.shape)
print(predictions)
predictions[predictions == -1] = 0
predictions = np.reshape(predictions,(eval_data.X.shape[0],1))
print(predictions.shape)
print(predictions)
eval_out = np.hstack((eval_ids,predictions))
print(eval_out.shape)
print(eval_out)
eval_df = pd.DataFrame(data = eval_out,index = None,columns=['example_id','label'])
save_to_path = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decis
eval_df.to_csv(path_or_buf=save_to_path,index=False)
```

```
(5250,)
[ 1. -1.  1. ...  1. -1. -1.]
(5250, 1)
[[1.]
 [0.]
 [1.]
 ...
 [1.]
 [0.]
 [0.]]
(5250, 2)
[[0.000e+00 1.000e+00]
 [1.000e+00 0.000e+00]
 [2.000e+00 1.000e+00]
 ...
 [5.247e+03 1.000e+00]
 [5.248e+03 0.000e+00]
 [5.249e+03 0.000e+00]]
```

Save weights for later use

```
outfile = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/
np.save(outfile, W)
```