Neural Networks and Deep Learning (CSCI 5922)

Spring 2020

Name: Nicholas Renninger

Goal

The goal of this assignment is to introduce neural networks in terms of ideas you are already familiar with: linear regression and classification

Dataset

You are given a dataset with 2 input variables (x_1, x_2) and an output variable (y).

```
from matplotlib import pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
import os
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('retina')

# Load data
data = np.loadtxt(os.path.join('data', 'assign1_data.txt'),  delimite
predictors = data[:, :2]
targets = data[:, 2].reshape((-1, 1))
target_class_labels = data[:, 3].reshape((-1, 1))

numExamples = len(predictors)
```

executed in 1.10s, finished 11:06:59 2020-02-10

Part 1

Write a program to find the exact least squares solution to $y = w_1 x_1 + w_2 x_2 + b$ for the above dataset, using the normal equation.

Complete the following function below and use it to answer questions (A) and (B).

Note: Please do not change the interface of the given function.

```
▸ def least_squares(X, y):↔

▸ def getFeaturesWithBias(X):↔

▸ def computePointwiseQuadraticLoss(y, y_hat):↔
```

executed in 5ms, finished 11:06:59 2020-02-10

(A) Report the values of w_1, w_2, and b.

```
weights = least_squares(predictors, targets)
regressed_targets = getFeaturesWithBias(predictors) @ weights

print('w_1:', weights[0])
print('w_2:', weights[1])
print('b:', weights[2])
```

executed in 7ms, finished 11:06:59 2020-02-10

```
w_1: [-2.0442426]
w_2: [3.99686017]
b: [-0.92429081]
```

$$w_1 = -2.04424$$

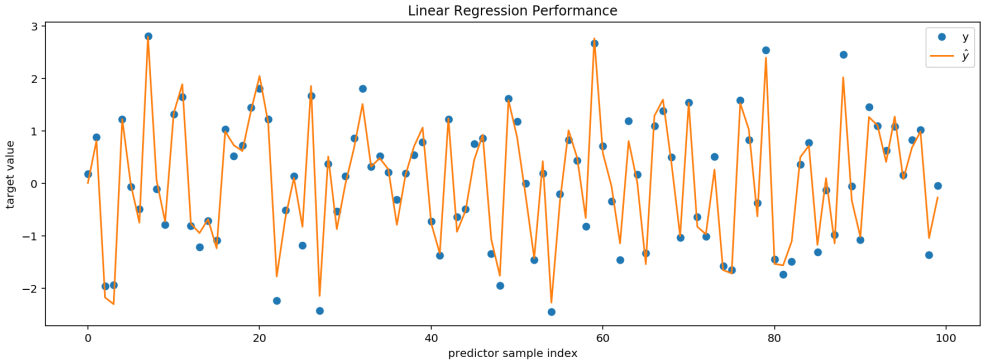
$$w_2 = 3.99686$$

$$b = -0.92429$$

Now, we can examine the fitting performance of our linear regression model. Below, we see how the regressed target values compare to the observed target values for each predictor. They match quite well, as is expected on the test set.

```
def plot_regression_performance(predictor_indices, targets, regressed
predictor_indices = range(0, numExamples)
plot_regression_performance(predictor_indices, targets, regressed_ta
```

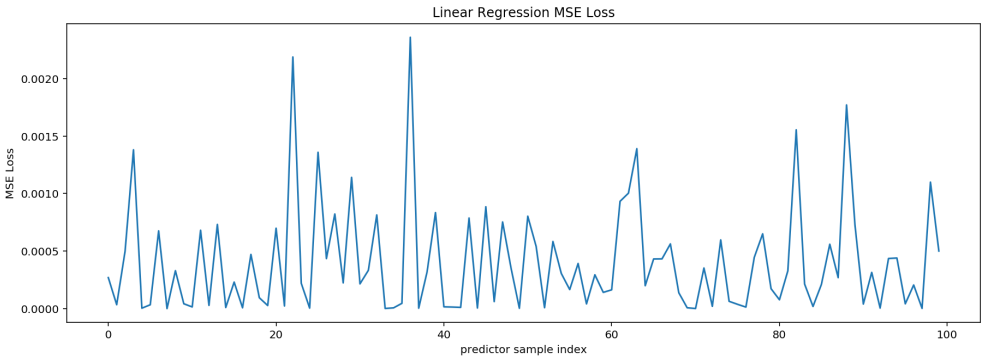
executed in 268ms, finished 11:07:38 2020-02-10



Here we investigate the loss term as a function of each predictor. It is as expected, with the loss generally being quite low, except for some outliers that don't fit the linear model well.

```
def plot_loss(indices, lossData, xlabel, ylabel, **kwargs):↔
loss = computePointwiseQuadraticLoss(targets, regressed_targets)
plot_loss(predictor_indices, [loss],
          xlabel='predictor sample index',
          ylabel='MSE Loss')
```

executed in 272ms, finished 11:06:59 2020-02-10



(B) What function or method did you use to find the least-squares solution?

The least-squares solution is the result of optimizing:

$$X \cdot \mathbf{w} = Y$$

where $\hat{\mathbf{w}}$ is the estimated weight vector, Y is the target vector, and $\hat{Y}$ is the regressed target vector. We optimize our quadratic loss to find $\hat{\mathbf{w}}$:

$$\begin{aligned}\hat{\mathbf{w}} &= \mathbf{w} \mid (Y - \hat{Y})^T \cdot (Y - \hat{Y}) = 0 \\ &= \mathbf{w} \mid (Y - X \cdot \mathbf{w})^T \cdot (Y - X \cdot \mathbf{w}) = 0\end{aligned}$$

The solution, assuming a linearly independent columns of X, can be robustly computed with the Moore-Penrose psuedoinverse of the predictor matrix X:

$$X^+ = \left(X^T X\right)^{-1} X^T$$

Thus, the loss-minimizing weight vector $\hat{\mathbf{w}}$ is computed as:

$$\hat{\mathbf{w}} = X^+ Y$$

Part 2

Implement linear regression of y on X via first-order optimization of the least-squares objective. Write a program that determines the coefficients {w1,w2,b}. Implement stochastic gradient descent, batch gradient descent, and mini-batch gradient descent. You will need to experiment with updating rules, step sizes (i.e. learning rates), stopping criteria, etc. Experiment to find settings that lead to solutions with the fewest number of sweeps through the data.

Complete the following functions below and use them to answer questions (A), (B) and (C). You may find the shuffle function from scikit-learn useful.

Use the following hyperparameters:

```
Learning rates = [0.001, 0.05, 0.01, 0.05, 0.1, 0.3]
```

```
MaxIter = [10, 50, 100, 500, 1000, 5000, 10000, 25000, 50000]
```

Note: Please do not change the interface of the given functions.

```
from sklearn.utils import shuffle

def compute_MSE_loss(y, y_hat):↔

def get_batch_indices(curr_idx, batch_size, num_batches, batch_remai

def least_squares_loss_gradient(X, y, w):↔

def online_epoch(X, y, w, alpha, *batch_size):↔

def batch_update(X, y, w, alpha, *batch_size):↔

def mini_batch_update(X, y, w, alpha, batch_size):↔

def least_squares_grad_desc(X, y, maxIter, alpha, update, **kwargs):
```

executed in 66ms, finished 11:06:59 2020-02-10

(A) Report the values of w_1, w_2, and b.

```
# update options are:↔
```

executed in 111ms, finished 11:07:00 2020-02-10

```
Epoch:  0 Loss:  1.2460606286988651
Epoch:  100 Loss:  0.0402896051330315
Max Epochs:  107 Min Loss:  0.040341341788431344
w_1: [-2.0372031]
w_2: [3.9789449]
b: [-0.92883046]
```

(B) *What settings worked well for you: online vs. batch vs. minibatch? What step size? How did you decide to terminate?*

The best settings I found were:

- `update` : `'minibatch'`
- `alpha` : 0.01
- `maxIter` : 1000
- `relTol` : 1e-6

These settings were empirically determined by empirical optimization. Raising `maxIter` much more than 1000 would result in the loss being unchanged for almost the entire calculation. Thus, the relative tolerance was implemented to stop the iteration early, once the loss between successive gradient descent iterations changed by less than `relTol` . Thus, the gradient descent typically finished before completing `maxIter` epochs.

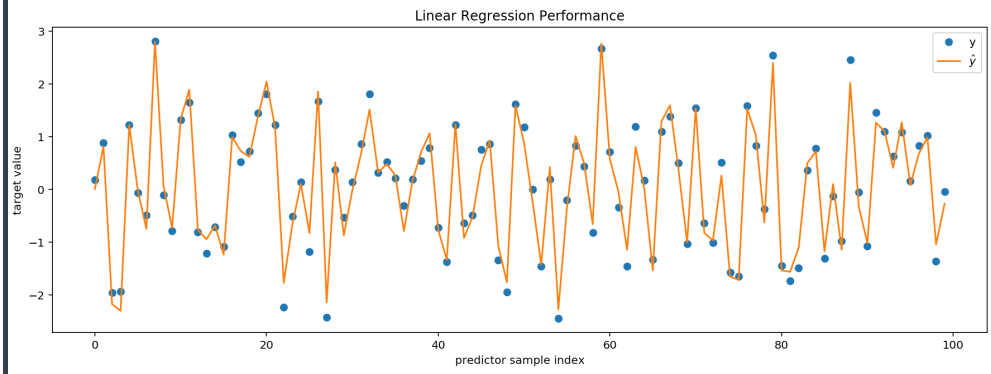
I found that without gradient momentum, making `alpha` smaller than 0.01 caused extremely slow convergence, except for the `batch` update, which required about an order of magnitude smaller learning rate.

The selection of ***mini-batch*** made sense, as the data was found to fit a linear model quite well, so ***online*** and ***mini-batch*** both perform much better than ***batch*** , as the statistics are captured by small subsets of the data (as seen in the loss convergence plot below). The selection of ***mini-batch*** over ***online*** was done, as while ***online*** typically converged in fewer epochs, its poor per-epoch computation time (due to the large number of gradient evaluations) made it take slightly more time to converge to the loss tolerance.

```
maxEpochs = 100

weights, lossPerEpoch = least_squares_grad_desc(predictors, targets,
loss_SGD = lossPerEpoch

weights, lossPerEpoch = least_squares_grad_desc(predictors, targets,
loss_batch = lossPerEpoch

weights, lossPerEpoch = least_squares_grad_desc(predictors, targets,
loss_miniBatch = lossPerEpoch

plot_regression_performance(predictor_indices, targets, regressed_ta
```

executed in 1.22s, finished 11:07:01 2020-02-10

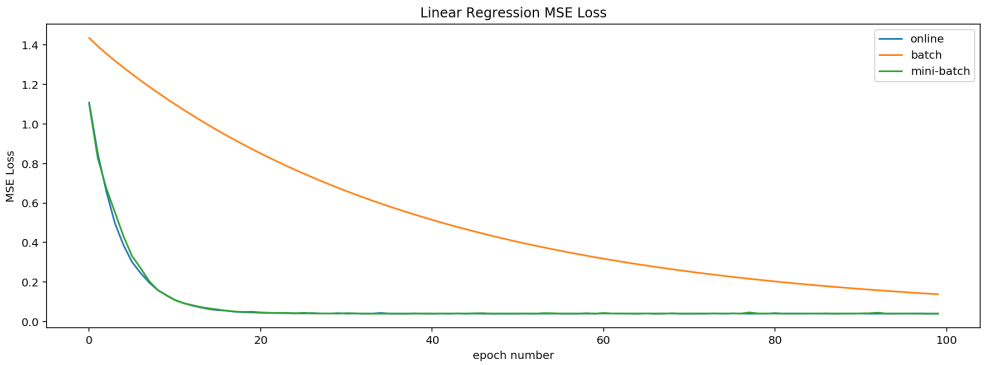
```
Epoch:  0 Loss:  1.1076776102445716
Max Epochs:  100 Min Loss:  0.040247360299446575
Epoch:  0 Loss:  1.4351012327555464
Max Epochs:  100 Min Loss:  0.13854167138443022
Epoch:  0 Loss:  1.0994042694717765
Max Epochs:  100 Min Loss:  0.040548638708831825
```



(C) Make a graph of error on the entire data set as a function of epoch. An epoch is a complete sweep through all the data (which is one iteration for full-batch gradient descent).

```
test_epoch_indices = range(0, len(lossPerEpoch))
plot_loss(test_epoch_indices, [loss_SGD, loss_batch, loss_miniBatch],
          xlabel='epoch number',
          ylabel='MSE Loss',
          legendStrings=('online', 'batch', 'mini-batch'))
```

executed in 227ms, finished 11:07:01 2020-02-10



Part 3

The data set from a regression problem can be converted into a classification problem simply by using the sign of (+ or -) as representing one of two classes. In the data set used in Part 1 and 2, you'll see the variable z that represents this binary (0 or 1) class.

Use the perceptron learning rule to solve for the coefficients $\{w_1, w_2, b\}$ of this classification problem.

Two warnings: First, your solution to Part 3 should require only a few lines of code changed from the code you wrote for Part 2. Second, the Perceptron algorithm will not converge if there is no exact solution to the training data. It will jitter among coefficients that all yield roughly equally good solutions.

Complete the following functions below and use them to answer questions (A) and (B).

Note: Please do not change the interface of the given functions.

```
def perceptron_update(X, y, w, alpha):↔

def perceptron(X, y, maxIter, alpha):↔
```

executed in 8ms, finished 11:07:01 2020-02-10

(A) Report the values of coefficients w_1, w_2, and b.

```
# need negative classes to be -1
perceptron_labels = [-1 if label == 0 else 1 for label in target_cla

maxIter = 1e4
(weights,
 incorrect,
 accuracyPerEpoch) = perceptron(predictors, perceptron_labels,
                                maxIter=maxIter, alpha=1.0)

print('w_1:', weights[0])
print('w_2:', weights[1])
print('b:', weights[2])
```

executed in 4.21s, finished 11:07:05 2020-02-10

```
Epoch:  0 accuracy:  0.76
Epoch:  100 accuracy:  0.94
Epoch:  200 accuracy:  0.94
Epoch:  300 accuracy:  0.92
Epoch:  400 accuracy:  0.94
Epoch:  500 accuracy:  0.92
Epoch:  600 accuracy:  0.92
Epoch:  700 accuracy:  0.94
Epoch:  800 accuracy:  0.92
Epoch:  900 accuracy:  0.94
Epoch:  1000 accuracy:  0.92
Epoch:  1100 accuracy:  0.94
Epoch:  1200 accuracy:  0.92
Epoch:  1300 accuracy:  0.92
Epoch:  1400 accuracy:  0.95
Epoch:  1500 accuracy:  0.95
Epoch:  1600 accuracy:  0.94
Epoch:  1700 accuracy:  0.94
Epoch:  1800 accuracy:  0.95
Epoch:  1900 accuracy:  0.95
Epoch:  2000 accuracy:  0.94
Epoch:  2100 accuracy:  0.95
Epoch:  2200 accuracy:  0.95
Epoch:  2300 accuracy:  0.92
Epoch:  2400 accuracy:  0.94
Epoch:  2500 accuracy:  0.95
Epoch:  2600 accuracy:  0.92
Epoch:  2700 accuracy:  0.94
Epoch:  2800 accuracy:  0.95
Epoch:  2900 accuracy:  0.92
Epoch:  3000 accuracy:  0.95
Epoch:  3100 accuracy:  0.92
Epoch:  3200 accuracy:  0.92
Epoch:  3300 accuracy:  0.95
Epoch:  3400 accuracy:  0.92
Epoch:  3500 accuracy:  0.95
Epoch:  3600 accuracy:  0.95
Epoch:  3700 accuracy:  0.92
Epoch:  3800 accuracy:  0.94
```

```
Epoch:  3900 accuracy:  0.95
Epoch:  4000 accuracy:  0.92
Epoch:  4100 accuracy:  0.94
Epoch:  4200 accuracy:  0.92
Epoch:  4300 accuracy:  0.92
Epoch:  4400 accuracy:  0.94
Epoch:  4500 accuracy:  0.92
Epoch:  4600 accuracy:  0.92
Epoch:  4700 accuracy:  0.94
Epoch:  4800 accuracy:  0.95
Epoch:  4900 accuracy:  0.92
Epoch:  5000 accuracy:  0.94
Epoch:  5100 accuracy:  0.95
Epoch:  5200 accuracy:  0.95
Epoch:  5300 accuracy:  0.94
Epoch:  5400 accuracy:  0.92
Epoch:  5500 accuracy:  0.94
Epoch:  5600 accuracy:  0.94
Epoch:  5700 accuracy:  0.92
Epoch:  5800 accuracy:  0.94
Epoch:  5900 accuracy:  0.92
Epoch:  6000 accuracy:  0.92
Epoch:  6100 accuracy:  0.94
Epoch:  6200 accuracy:  0.95
Epoch:  6300 accuracy:  0.95
Epoch:  6400 accuracy:  0.95
Epoch:  6500 accuracy:  0.94
Epoch:  6600 accuracy:  0.95
Epoch:  6700 accuracy:  0.94
Epoch:  6800 accuracy:  0.95
Epoch:  6900 accuracy:  0.92
Epoch:  7000 accuracy:  0.94
Epoch:  7100 accuracy:  0.95
Epoch:  7200 accuracy:  0.92
Epoch:  7300 accuracy:  0.92
Epoch:  7400 accuracy:  0.92
Epoch:  7500 accuracy:  0.95
Epoch:  7600 accuracy:  0.92
Epoch:  7700 accuracy:  0.92
Epoch:  7800 accuracy:  0.95
Epoch:  7900 accuracy:  0.92
Epoch:  8000 accuracy:  0.94
Epoch:  8100 accuracy:  0.94
Epoch:  8200 accuracy:  0.95
Epoch:  8300 accuracy:  0.92
Epoch:  8400 accuracy:  0.94
Epoch:  8500 accuracy:  0.95
Epoch:  8600 accuracy:  0.95
Epoch:  8700 accuracy:  0.92
Epoch:  8800 accuracy:  0.92
Epoch:  8900 accuracy:  0.94
Epoch:  9000 accuracy:  0.92
Epoch:  9100 accuracy:  0.92
Epoch:  9200 accuracy:  0.94
Epoch:  9300 accuracy:  0.95
Epoch:  9400 accuracy:  0.92
Epoch:  9500 accuracy:  0.95
```

```
Epoch:  9600 accuracy:  0.94
Epoch:  9700 accuracy:  0.95
Epoch:  9800 accuracy:  0.95
Epoch:  9900 accuracy:  0.94
Max Epochs:  10000 Max Accuracy:  0.92
w_1: [-23.0966]
w_2: [38.8621]
b: [-6.]
```

$$w_1 = -23.0966$$

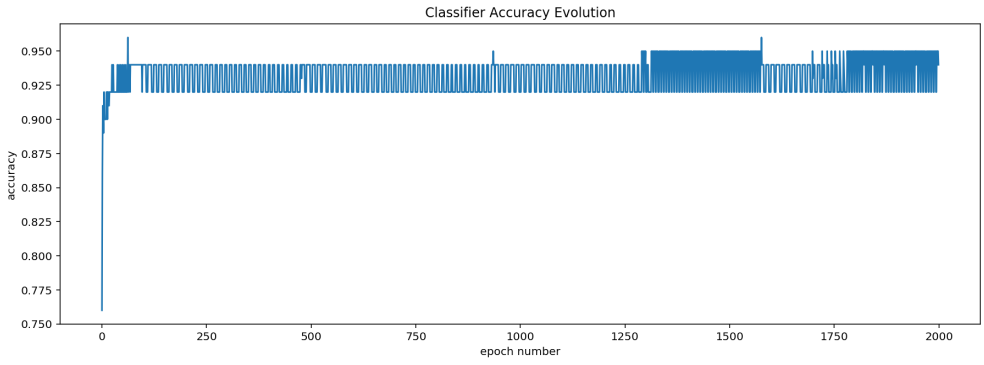
$$w_2 = 38.8621$$

$$b = -6$$

(B) Make a graph of the accuracy (% correct classification) on the training set as a function of epoch.

```
# if lots of epochs were used, only use part of them
maxEpochsToPlot = 500
if maxIter > maxEpochsToPlot:
    maxIter = maxEpochsToPlot

maxIter = int(maxIter)
test_epoch_indices = range(maxIter)

plot_loss(test_epoch_indices, [accuracyPerEpoch[0:maxIter]],
          xlabel='epoch number',
          ylabel='accuracy',
          title='Classifier Accuracy Evolution')
```

executed in 257ms, finished 11:07:05 2020-02-10



Part 4

In machine learning, we really want to train a model based on some data and then expect the model to do well on "out of sample" data. Try this with the code you wrote for Part 3: Train the model on the first {5, 10, 25, 50, 75} examples in the data set and

test the model on the final 25 examples.

Complete the following function below and use it to answer (A).

Note: Please do not change the interface of the given function.

```
def classify(X, y, w):↔
```

executed in 4ms, finished 11:07:05 2020-02-10

Now, we will vary the size of the training set and collect the accuracies of the models associated with each of these training-test splits and save the test accuracy of the model to test model generalization.

```
def studyPerceptronPerformance(predictors, labels,↔

trainSetSizes = [5, 10, 25, 50, 75]
testSetAccuracies = studyPerceptronPerformance(predictors=predictors,
                                               labels=perceptron_labe
                                               trainSetSizes=trainSet
                                               testSetSize=25,
                                               maxIter=100, alpha=1.0
```

executed in 46ms, finished 11:07:06 2020-02-10

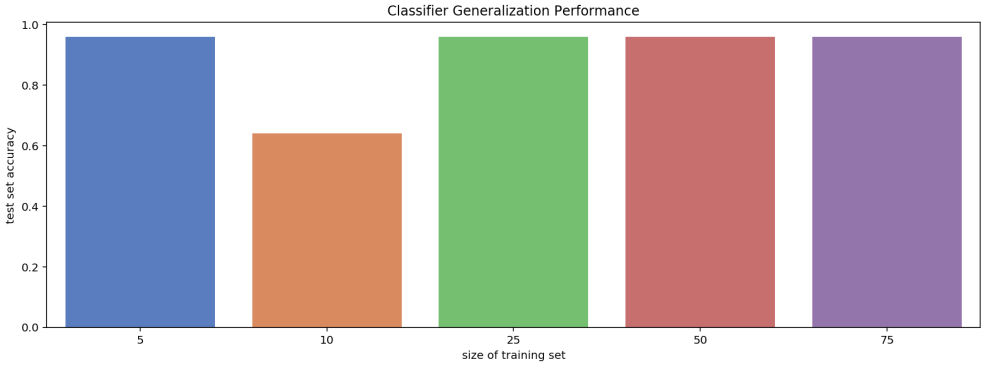
```
Epoch:  0 accuracy:  0.4
Max Epochs:  2 Max Accuracy:  1.0
test accuracy: 0.96
Epoch:  0 accuracy:  0.6
Max Epochs:  4 Max Accuracy:  1.0
test accuracy: 0.64
Epoch:  0 accuracy:  0.68
Max Epochs:  15 Max Accuracy:  1.0
test accuracy: 0.96
Epoch:  0 accuracy:  0.68
Max Epochs:  16 Max Accuracy:  1.0
test accuracy: 0.96
Epoch:  0 accuracy:  0.72
Max Epochs:  100 Max Accuracy:  0.92
test accuracy: 0.96
```

How does performance on the test set vary with the amount of training data? Make a bar graph showing performance for each of the different training set sizes.

Surprisingly, the performance of the classifier on the test set (i.e. generalization error) seems to be fairly independent of the training. This is again due to the data following a linear model quite well, allowing the learner to generalize well even seeing very few examples of the model. This excellent generalization performance is even more impressive given the complete lack of effort to introduce regularization to our classifier.

```
def plotGeneralizationPerformance(trainSetSizes, testSetAccuracies):

plotGeneralizationPerformance(trainSetSizes, testSetAccuracies)
```

executed in 236ms, finished 11:07:06 2020-02-10

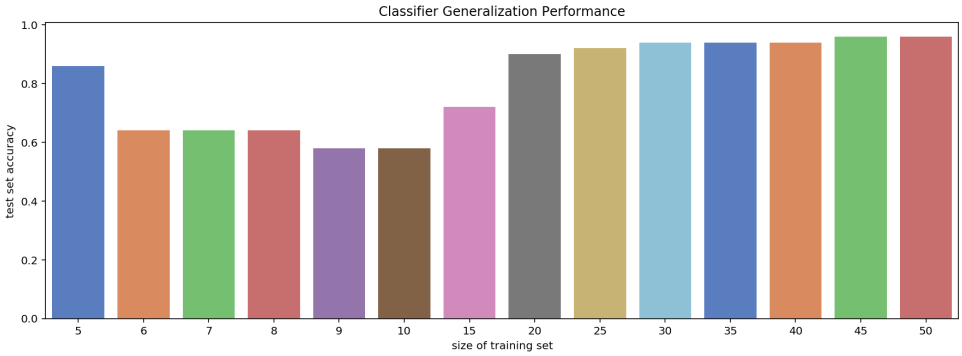


However, we can see that there are some noisy examples in the data set, as increasing the size of the test set leads to generalization performance more in line with our expectations. Below, we see that the classifier monotonically increases after seeing 10 examples. The first 5 examples seem to be more illustrative of the underlying linearly-separable data generating process, while the next 10 or diverge more heavily from this sort of model.

```
trainSetSizes = [5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50]
testSetAccuracies = studyPerceptronPerformance(predictors=predictors,
                                               labels=perceptron_labe
                                               trainSetSizes=trainSet
                                               testSetSize=50,
                                               maxIter=100, alpha=1.0
plotGeneralizationPerformance(trainSetSizes, testSetAccuracies)
```

executed in 358ms, finished 11:07:06 2020-02-10

```
Epoch:  0 accuracy:  0.4
Max Epochs:  2 Max Accuracy:  1.0
test accuracy: 0.86
Epoch:  0 accuracy:  0.3333333333333333
Max Epochs:  3 Max Accuracy:  1.0
test accuracy: 0.64
Epoch:  0 accuracy:  0.42857142857142855
Max Epochs:  3 Max Accuracy:  1.0
test accuracy: 0.64
Epoch:  0 accuracy:  0.5
Max Epochs:  3 Max Accuracy:  1.0
test accuracy: 0.64
Epoch:  0 accuracy:  0.5555555555555556
Max Epochs:  4 Max Accuracy:  1.0
test accuracy: 0.58
Epoch:  0 accuracy:  0.6
Max Epochs:  4 Max Accuracy:  1.0
test accuracy: 0.58
Epoch:  0 accuracy:  0.6
Max Epochs:  5 Max Accuracy:  1.0
test accuracy: 0.72
Epoch:  0 accuracy:  0.65
Max Epochs:  84 Max Accuracy:  1.0
test accuracy: 0.9
Epoch:  0 accuracy:  0.68
Max Epochs:  15 Max Accuracy:  1.0
test accuracy: 0.92
Epoch:  0 accuracy:  0.6666666666666666
Max Epochs:  9 Max Accuracy:  1.0
test accuracy: 0.94
Epoch:  0 accuracy:  0.6857142857142857
Max Epochs:  23 Max Accuracy:  1.0
test accuracy: 0.94
Epoch:  0 accuracy:  0.7
Max Epochs:  22 Max Accuracy:  1.0
test accuracy: 0.94
Epoch:  0 accuracy:  0.6666666666666666
Max Epochs:  13 Max Accuracy:  1.0
test accuracy: 0.96
Epoch:  0 accuracy:  0.68
Max Epochs:  16 Max Accuracy:  1.0
test accuracy: 0.96
```


Classifier Generalization Performance
test set accuracy
size of training set
0.0
0.2
0.4
0.6
0.8
1.0
5
6
7
8
9
10
15
20
25
30
35
40
45
50