CS 461: Homework #1

Stats and Data

Problem 1.

Solution Let $X \sim \text{Bernoulli}(p)$. Then, $\bar{X_n}$ represents the average probability a person supports Candidate A from a sample size n. The problem asks us to determine n such that we are 95% confident that $\bar{X_n}$ is within an error of 0.01 to p. Rewrite the problem statement in terms of an inequality:

$$P(|\bar{X_n} - p| < 0.01) \geq 0.95$$

Negate the inequality. We get

$$P(|\bar{X_n} - p| > 0.01) \leq 0.05$$

Use Hoeffding's inequality. We find that

$$2e^{-2 \cdot (0.01)^2 \cdot n} = 0.05$$

Solving gives $n \geq 18444.4$. Thus, you should poll $\boxed{18445}$ people to be 95% confident that you know the value of p within an error of 0.01.

Problem 2.

Solution Follow the previous problem. We start with the inequality

$$P(|\bar{X_n} - p| > \epsilon) \leq 0.05$$

Again, apply Hoeffding's inequality. Since $n = 30$,, we have

$$2e^{-60\epsilon^2} = 0.05$$

Solving for ϵ, we get

$$\boxed{\epsilon \approx \pm 0.248}$$

This means that on average, our sample mean from $k = 30$ people will have an error of roughly 0.248.

Problem 3.

Solution

```python
import math
import numpy as np

size = 30 #size as per the problem
prob = 0.55 #probability of bernoulli distribution

samples = np.random.binomial(n = 1, p = prob, size = size) #generate 30 bernoulli trials
sample_mean = samples.mean() #take mean of them
print(sample_mean) #0.6
```

The above python code gave $\hat{p} = 0.6$. Evidently, this corresponds with the 95% confidence interval as 0.6 is within $\epsilon = 0.248$ range from the actual probability of 0.55.

Problem 4.

Solution The number of votes that Candidate A receives will take on a binomial distribution. Each person that votes in the election has a probability p of voting for Candidate A, which takes on a Bernoulli distribution. Since every person votes independently, the combination of N such Bernoulli distributions is a binomial distribution.

Problem 5.

Solution For a given $i, 0 \leq i \leq 1000$, the probability of i people voting for candidate A is

$$P(X = i) = \binom{1000}{i}(0.55)^i(0.45)^{1000-i}$$

The probability of candidate A winning the election (receiving a majority of the votes) is simply

$$\sum_{i=501}^{1000} P(X = i) = \sum_{i=501}^{1000} \binom{1000}{i}(0.55)^i(0.45)^{1000-i}$$

The code below does this calculation:

```python
def cdf(p):
    pb = 0
    for i in range(501, 1001):
        pb += math.comb(1000, i) * pow(p, i) * pow(1 - p, 1000 - i)
    return pb
print(cdf(0.55)) #0.99153
```

$$\boxed{\sum_{i=501}^{1000} P(X = i) \approx 0.9991534508}$$

Problem 6.

Solution Use $\hat{p}$ instead of p in the summation above. We get

$$\boxed{\sum_{i=501}^{1000} \binom{1000}{i}(0.6)^i(0.4)^{1000-i} \approx 0.9999999998985}$$

Problem 7.

Solution

```python
mu = sample_mean
sigma = math.sqrt((mu * (1 - mu)) / size)

avg, i = 0, 0
while i < 1000:
    clt = np.random.normal(mu, sigma, 1)
    if clt[0] >= 0 and clt[0] <= 1: #check if probability is between 0 and 1
        avg += cdf(clt[0]) #add the cdf with that probability
        i += 1
avg = avg / 1000 #take average of all probabilities across 1000 "trials"

print('Probability of Candidate A winning on CLT sampled p_hat: ', avg) #0.8743088330840642
```

The new estimate for the probability of candidate A winning is approximately 0.87431. Evidently, this is lower than the result we found in question 1.6. This should make sense. Sampling from the distribution $\mathcal{N}(0.6, 0.0894)$ is going to give some values that are less than 0.6, and some that are more. Taking this value

as p 1000 times will then give a result that is smaller than the value we got with just $\hat{p}$.

Bonus Question

Solution In the 1948 election of Thomas Dewey and Harry Tubman, the Chicago Daily Tribune had incorrectly stated that Dewey had won the election (this is where the infamous headline: DEWEY DEFEATS TRUMAN comes from). Indeed, Truman was the one that ended up winning the election with a margin of 49.6 percent of the popular vote compared to Dewey's 45.1 percent.

It was later discovered that the Tribune's poll had forecasted incorrectly as the sample of Americans they had polled did not represent the public opinion of America. The poll was conducted entirely over the phone. In the 1950s, wealthier families were more likely to have a phone and were also more likely to vote Republican. Hence, by selecting a sample that was not random and was skewed significantly Republican, the poll incorrectly predicted that Dewey would win the election.

In the context of machine learning and statistics, this is often referred to as a sampling bias and is a very common issue when dealing with machine learning algorithms. Sampling biases happen when the data is selected in such a way that it does not truly reflect the entire population and there is some underlying distribution within the sample itself. In machine learning, we are presented with the task of constructing an accurate model from a large dataset. The task of building a model means that we need to understand the underlying distribution within the dataset. Oftentimes, this is very hard. Hence, in order for the model to have the highest chance of predicting all outcomes, the training and testing datasets must have the same distribution of outcomes that would be seen in the real world. Additionally, researchers and statisticians also use random sampling techniques such as cluster and stratified sampling to ensure that their data represents the population they are analyzing.

Regression Comparison

Please note that for each problem, I have provided a short code snippet that supports my explanations. This code snippet is intended to be standalone. Refer to prob2.py for the full code. Below is the generate_data function that is used throughout all code snippets.

```python
def generate_data(size, d):
    x = []
    x.append(np.random.normal(3, 1, size))
    x.append(np.random.normal(-2, 1, size))
    x.append(x[0] + 2 * x[1])
    x.append((x[1] + 2)**2)
    x.append(np.random.binomial(n=1, p=0.8, size=size))
    for _ in range(d - 5):
        x.append(np.random.normal(0, 0.1, size))

    def compute_y(x):
        y = 4 - 3 * x[0] * x[0] + x[2] - 0.01 * x[3] + x[1] * x[4] + np.random.normal(0, 0.1,
        ↪  len(x[0]))
        return y

    def transpose(temp):
        temp = np.array(temp)
        return temp.T

    xt = transpose(x)
    y = compute_y(x)

    return x, xt, y
```

Problem 1.

Solution Pick c to be the mean of the y values of the dataset. We would like c to give the minimum mean squared error for our dataset. Define the points $(y_1, \ldots, y_N)$ as a list of y values from a dataset of N rows. The mean squared error is defined as

$$f(c) = \sum_{i=1}^{N} (y_i - c)^2$$

where c represents the constant model. We would like to minimize $f(c)$ for all c. Take the derivative of $f(c)$. We get

$$f'(c) = -\frac{2}{N} \sum_{i=1}^{N} (y_i - c)$$

Setting it equal to 0, we get

$$-\frac{2}{N} \sum_{i=1}^{N} (y_i - c) = 0$$

$$\sum_{i=1}^{N} (y_i - c) = 0 \Rightarrow \sum_{i=1}^{N} y_i = Nc$$

$$\boxed{c = \frac{1}{N} \sum_{i=1}^{N} y_i}$$

Thus, we have shown that $c = \bar{y}$ is the best constant model. Intuitively, this makes sense as the mean encompasses all deviations of the y values of the dataset. The following code shows how to find the optimal constant model and the error that it had on training/testing data.

```python
def question1():
    x, xt, y = generate_data(10000, 10)
    _, tx, testy = generate_data(1000, 10)
    #constant model = the mean of y values (explained in pdf)
    optimal_c = np.mean(y)

    #function that computes the MSE of a given constant model
    def compute_mse_constant_model(y, c):
        err = 0
        for i in range(len(y)):
            err += (c - y[i]) ** 2
        err = err / len(y) #take average of squared sum
        return err

    #Find the errors of both training and testing
    training_error = compute_mse_constant_model(y, optimal_c)
    testing_error = compute_mse_constant_model(testy, optimal_c)

    #Print
    print('Optimal C: ', optimal_c)
    print('Training Error: ', training_error)
    print('Testing Error: ', testing_error)

    return optimal_c, training_error, testing_error
question1()
#Optimal C:  -28.688845246408892
#Training Error:  323.62510754827 39
#Testing Error:  319.58543388909504
```

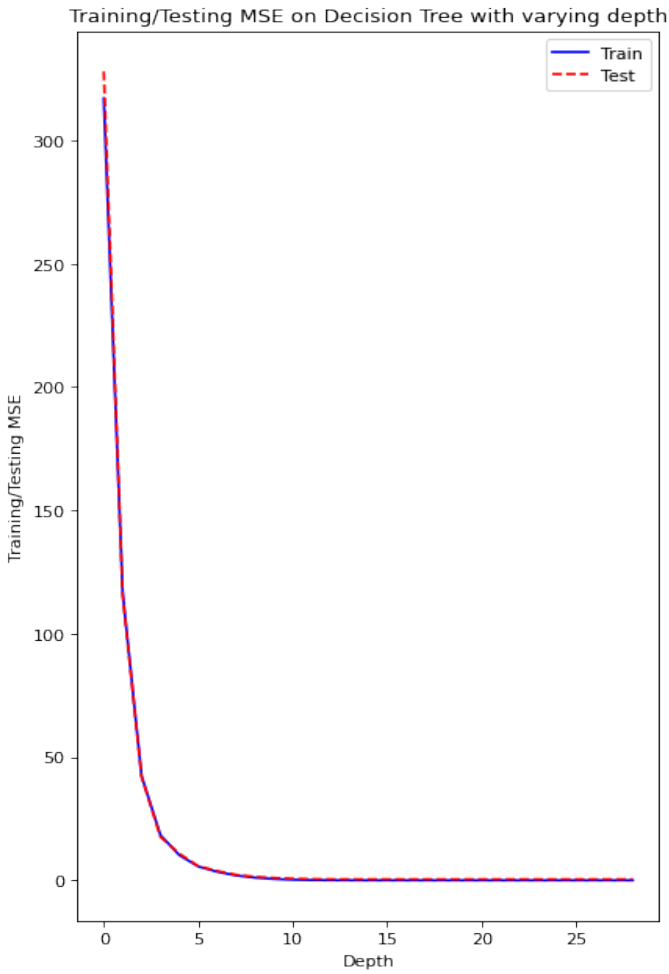
The value of c that we should pick is approximately -28.533 from the code. The training error for this constant model is approximately 316.961, and the testing error is approximately 327.851. The training error seems to be larger than the testing error.
While this may be surprising, it is to be expected. With a constant model, the training and testing errors are somewhat random as the model does not learn anything about the features. Hence, merely predicting the average of the y values is a representation of an undertrained model, giving us a higher training error than a testing error. I have run this model multiple times, however. Sometimes, I have seen training being much lower than testing, solidifying that the errors with the constant model are somewhat arbitrary and hard to predict exactly. The error has been consistently greater than 300 for both training and testing error though.
Lastly, the value of d does not matter as the additional features that we add to the dataset do not influence the value of Y. Therefore, the value of d will not change the accuracy of the constant model.

Problem 2.

Solution The following code fits a dataset of $d = 10$ to decision trees of varying depths (from 1 to 50). The minimum sample size for all of these decision trees is constant at 1, meaning that pruning is not influenced by the minimum sample size.

```
def compute_mse_dt(x, y, dt):
    err = 0 #error
    for i in range(len(y)):
        yp = DecisionTree.predict(dt, x[i]) #get prediction from decision tree
        err += (abs(yp - y[i]) ** 2) #add squared error
    err = err / len(y) #take mean of squared error
    return err

#Question 2
def question2():
    #Error lists -- used for building the graphs
    train_errors = []
    test_errors = []

    #As we mentioned above, set the appropriate variables for our function
    x = x_train10
    y = y10

    tx = tx_test10
    testy = testy10

    xt = tx_train10

    for i in range(1, 30):
        print('Iteration: ', i)
        #provide copies of the data (we want to use the same dataset on each tree)
        dt = DecisionTree(copy.deepcopy(x), copy.deepcopy(y), 1, i, 1)

        #compute errors
        training_err = compute_mse_dt(xt, y, dt)
        test_err = compute_mse_dt(tx, testy, dt)

        train_errors.append(training_err)
        test_errors.append(test_err)

    #build plot (can be seen in pdf)
    plt.figure(figsize = (6, 10), dpi = 80)
    plt.title('Training/Testing MSE on Decision Tree with varying depth')
```

```
    plt.plot(train_errors, '-b', label = 'Train')
    plt.plot(test_errors, '--r', label = 'Test')
    plt.xlabel('Depth')
    plt.ylabel('Training/Testing MSE')
    plt.legend()
    plt.show()

    return train_errors, test_errors

train_2, test_2 = question2() #shows the graph
print('Testing Errors for increasing maximum depth: ')
for i in range(len(train_2)): #prints the training/testing errors
    print(i + 1, ' ', train_2[i], ' ', test_2[i])
```



```
Testing Errors for increasing maximum depth:
1   316.9608408641263   327.8508124523598
2   116.77375547488393   115.66207026487284
3   42.36475127951115   41.91968593997631
4   18.301632561637263   17.602661360664868
5   10.19400592332933   10.801180467540311
6   5.617318041505871   5.76579717169987
7   3.45344826674579   3.8220094782070007
8   1.9687266197671491   2.2359071286072356
9   1.0776327786282682   1.4711592271735041
10   0.538858286079536 1   0.9897468068344495
11   0.25050377355723896   0.6979014994086987
12   0.10865385257161818   0.5278042384625765
13   0.04433060779949975   0.4638619635501323
14   0.016135269488133937   0.45109585187851364
15   0.005284454438529201   0.44734596663113774
16   0.0015794552740927977   0.44522788487159115
17   0.000395199032619274   0.443845674114742
18   6.974724927684455e-05   0.44385202621248454
19   7.784773524292967e-06   0.44408875167111983
20   8.936665245430212e-07   0.44400498140472966
21   1.2304484629633835e-12   0.4439596592084996
22   0.0   0.4439596592084996
23   0.0   0.4439596592084996
24   0.0   0.4439596592084996
25   0.0   0.4439596592084996
26   0.0   0.4439596592084996
27   0.0   0.4439596592084996
28   0.0   0.4439596592084996
29   0.0   0.4439596592084996
```

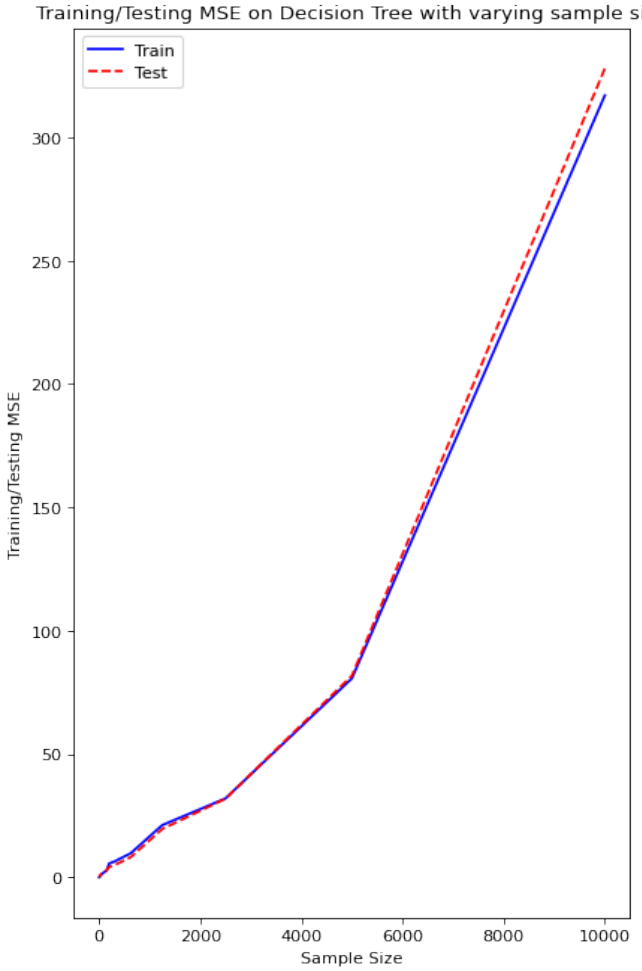
The optimal depth for the decision tree seems to be around 15. We see this through the second image, which shows the training and testing errors as the maximum depth allowed increases. The testing error reaches its minimum of 0.4438 at depth 17, indicating that this is the best performing decision tree. The training error at this depth is roughly 0.0003. Observe that beyond this depth, the testing error increases marginally and the training error drops to 0, showing that the model begins to overfit the dataset beyond this optimal depth.
The performance of the decision tree with optimal depth is substantially better than the constant model – note that the decision tree with a depth of 1 is the same as the constant model as if there are no splits within the dataset, then we default to predicting the average of all y values in the dataset.

Problem 3.

Solution The following code fits a dataset of $d = 10$ to decision trees of varying minimum sample sizes. The depths for all of these decision trees is constant at 150, meaning that depth is not considered while pruning the tree.

```python
#Question 3
def question3():
    #Lists for building error graphs
    train_errors = []
    test_errors = []

    #Again set the variables as appropriate
    x = x_train10
    y = y10

    tx = tx_test10
    testy = testy10

    xt = tx_train10

    #set of sample sizes that we will build decision trees out of
    #note that these are a set of decreasing and somewhat random numbers (more explanation in
    ↪  pdf)
```

```
    sample_sizes = [10000, 5000, 2500, 1250, 625, 300, 200, 150, 100, 50, 25, 20, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

    #generate dataset
    #x and y are used for the decision tree
    #xt is transpose of training dataset
    #tx (transpose of test dataset) and testy are a part of testing dataset
    iteration = 0
    for sample_size in sample_sizes: #iterate through each sample size
        #build decision tree with copies of dataset
        iteration += 1 #keep track of iteration
        print('Iteration: ', iteration)
        dt = DecisionTree(copy.deepcopy(x), copy.deepcopy(y), 1, 150, sample_size)

        #compute and store errors
        training_err = compute_mse_dt(xt, y, dt)
        test_err = compute_mse_dt(tx, testy, dt)
        train_errors.append(training_err)
        test_errors.append(test_err)

    #build plots
    plt.figure(figsize = (6, 10), dpi=80)
    plt.title('Training/Testing MSE on Decision Tree with varying sample size')
    plt.plot(sample_sizes, train_errors, '-b', label = 'Train')
    plt.plot(sample_sizes, test_errors, '--r', label = 'Test')
    plt.xlabel('Sample Size')
    plt.ylabel('Training/Testing MSE')
    plt.legend()
    plt.show()

    return train_errors, test_errors

train_3, test_3 = question3() #call function
for i in range(len(train_3)): #printing
    print(train_3[i], ' ', test_3[i])
```



Testing/Training Error on different sample sizes

10000	316.9608408641263	327.8508124523598
5000	80.64769017908375	81.75894601269162
2500	31.961880391419598	31.84309544127558
1250	21.14929041132909	19.567111999993518
625	9.732966250058272	8.128676434549385
300	6.3492310103585075	4.890940186363673
200	5.626507877240371	4.168813643847253
150	2.680953164859119	2.8842046270716732
100	2.0635777894382046	2.2158829576343986
50	1.2103675350166756	1.429296250155443
25	0.48724544663952135	0.7103019288807929
20	0.39034600185220447	0.5963078942587304
15	0.3032476226707787	0.4849140612747549
14	0.2756713815308897	0.47866622278138526
13	0.25311397104230066	0.4609138519261583
12	0.2237879230185075	0.45320117543655314
11	0.19966071003625957	0.43263572353917157
10	0.173587873585849	0.428633534981996
9	0.13377456763767248	0.4135540577283883
8	0.11479469146173649	0.4252220913771773
7	0.09043660927817611	0.3874590399601913
6	0.07191434959186309	0.37563018490266514
5	0.05443858800724401	0.4009819919867448
4	0.03452170930274354	0.39654438869337094
3	0.017473960570130672	0.41278962327558183
2	0.0067945187007445096	0.4417406744425394
1	0.0	0.4439596592084996

As before, the above shows the graph of the training/testing error and a chart with the training/testing error for different sample sizes. The graph shows that at high sample sizes, there is obviously high training/testing error, and as the sample size decreases, the error decreases. The optimal sample size threshold that minimizes the error is seen at the sample size 6 with a training error of approximately 0.072 and a testing error of approximately 0.3756. Again, the performance of the decision tree with optimal sample size threshold is substantially better than the constant model for the same reasons as before.

Problem 4.

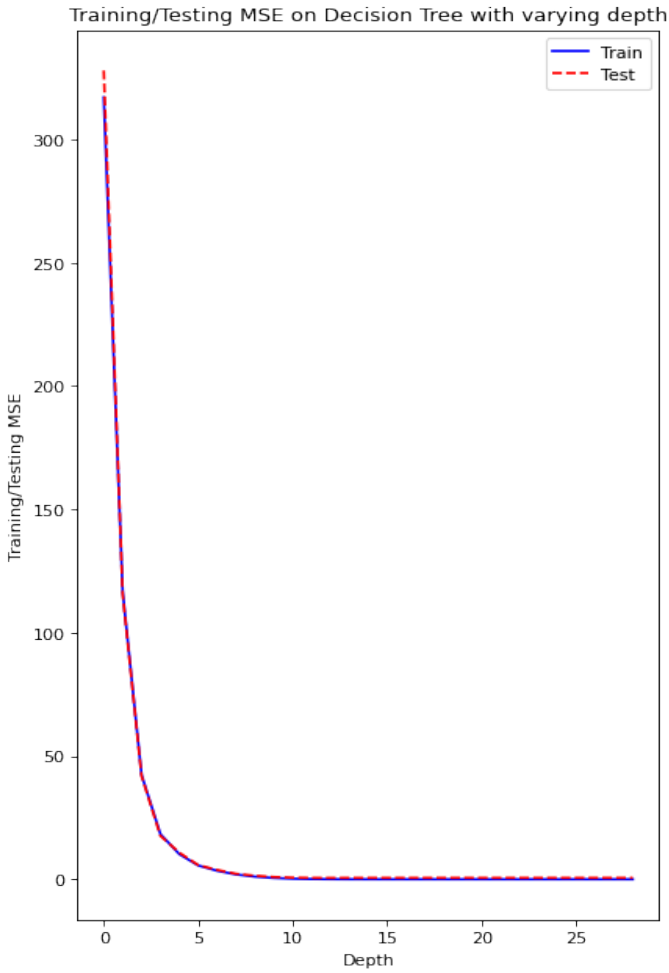
Solution From a high level, increasing the maximum possible depth and the minimum sample size of a node should have very similar effects on the growth of a decision tree. As the minimum sample size decreases, the decision tree will naturally grow deeper to split data within each node. Similarly, increasing maximum possible depth will lead to each leaf node of the tree to have a smaller sample size. Thus, as we see from the testing error from questions 2 and 3, it is important to truncate both by sample size and depth for any decision tree.

Now, while both sample size and depth prune a decision tree, the data from the prior questions indicate some differences between the two pruning techniques. It should be noted that from around depth 13, there isn't a significant difference in the testing error. In other words, the testing error converges rapidly when we truncate by depth, which is also seen in the graph for truncating by depth. However, when we truncate by sample size, the testing error never truly converges – it hovers around 0.38 and 0.44 from sample sizes 1 to 11. This means that truncating by depth is a stronger pruning technique than truncating by sample size as the rate of converge of the testing error is substantially faster with depth than sample size.

Lastly, I'd also like to point out that the minimum testing error across all decision trees was found when truncating by sample size. Thus, to wrap up my findings, if we compare the rate of testing error convergence, then truncating by depth is substantially better than truncating by sample size. However, if we are only looking at which pruning technique creates the smallest possible error, then truncating by sample size outperforms truncating by depth. All of these findings are supported in the graphs from the prior questions.

Problem 5.

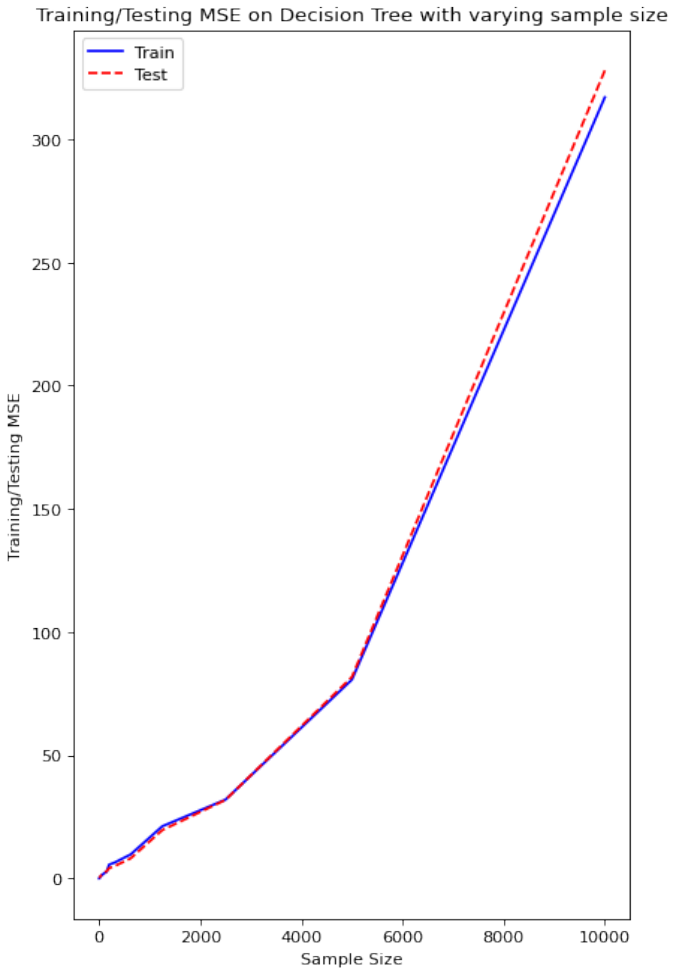
The code is very similar to questions 2 and 3 – the full code is on the python file (I didn't attach it here as there aren't any significant differences). The only difference between the prior questions and this question is just that the dataset has 50 features now rather than 10. The two figures below show the graph when truncating by depth (again where the minimum sample size is at 1) and the training/testing error.



Testing/Training Error on different depth

1	316.9608408641263	327.8508124523598
2	116.77375547488393	115.66207026487284
3	42.36475127951115	41.91968593997631
4	18.301632561637263	17.602661360664868
5	10.19400592332933	10.801180467540311
6	5.617318041505871	5.76579717169987
7	3.45344826674579	3.8220094782070007
8	1.9684274090179104	2.2359071286072356
9	1.075067626275039	1.4711316297934909
10	0.536364351073642	0.9342404422134372
11	0.24868951839817874	0.6836020936518177
12	0.10522713198941014	0.5846356692323268
13	0.04160980442573921	0.5653284165383163
14	0.014227174982224	0.5832441136530899
15	0.004075581781068561	0.5883510932593724
16	0.0009515536818286948	0.5924676000939841
17	0.0001652857422564806	0.5939564581568467
18	2.596452691906264e-05	0.5943924586330853
19	2.9175280917898916e-06	0.5947443084674715
20	2.8930251247767135e-07	0.5947952668554597
21	4.043174164085218e-08	0.5948021008733058
22	6.876638470564277e-10	0.5948023236964005
23	1.2304484629633835e-12	0.5948023236964005
24	0.0	0.5948023236964005
25	0.0	0.5948023236964005
26	0.0	0.5948023236964005
27	0.0	0.5948023236964005
28	0.0	0.5948023236964005
29	0.0	0.5948023236964005

The two figures below show the graph when truncating by sample size; again, these are very similar to the graphs where the dataset had only 10 features.



```
Testing/Training Error on different sample sizes
10000   316.9608408641263   327.8508124523598
5000   80.64769017908375   81.75894601269162
2500   31.961880391419598   31.84309544127558
1250   21.14929041132909   19.567111999993518
625   9.732966250058272   8.128676434549385
300   6.3492310103585075   4.890940186363673
200   5.626507877240371   4.168813643847253
150   2.680953164859119   2.8842046270716732
100   2.0635777894382046   2.2158829576343986
50   1.2103675350166756   1.429296250155443
25   0.4874306597645412   0.711508074777081
20   0.3905286671847874   0.603424247948223
15   0.3023087932276454   0.5169083358236971
14   0.27383909904488185   0.515441125161942
13   0.25177075880709715   0.4998943716184985
12   0.22090254764686984   0.5040136678756877
11   0.19682440103698753   0.5019431864325975
10   0.17551904194980125   0.4984347879003914
9   0.13163806177461837   0.49140997297360633
8   0.11104783250053755   0.5215930079427815
7   0.08321824460384293   0.5307383300968601
6   0.06305698781954848   0.5329167080161376
5   0.04715932416660075   0.534383359385249
4   0.027672318935349268   0.5561876903585746
3   0.015074403469852985   0.5658268452403195
2   0.004977914697610046   0.5817181998401258
1   0.0   0.5948023236964005
```

The four graphs above show the experiment repeated with the same dataset with $d = 50$ now (meaning that the first few questions used this dataset, but just the first 10 features). The optimal depth is now 13 with a training error of 0.04161 and a testing error of 0.5653. The optimal sample size is now 9 with a training error of 0.13163 and a testing error of 0.491409. All decision trees that were trained on this dataset had higher errors than the decision trees that were trained on the $d = 10$ dataset. This is to be expected as the 40 added features had no relevance to the y value.

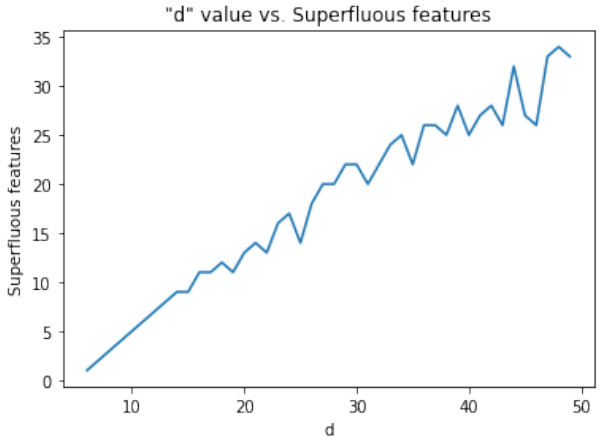
The optimal thresholds for both depth and sample size decreased compared to the previous decision trees. This also makes sense. The greater the depth threshold or the smaller the sample size threshold, the more vulnerable the model is to overfitting the dataset. Considering that 40 features are superfluous in this dataset, it is better if the model has a smaller depth threshold or a larger sample size threshold (compared with the $d = 10$ dataset) as it leads to more generalization, which gives us higher accuracy and a more powerful model.

Problem 6.

Solution

```python
#Question 6
def question6():
    sf_features = [] #hold the superfluous features
    indices = [] #hold the indices (for building the graph)
    for i in range(6, 50): #loop through from d = 6 to d = 50
        indices.append(i)
        x, _, y = generate_data(10000, i) #build new dataset
        #print('Number of features: ', i)
        dt = DecisionTree(copy.deepcopy(x), copy.deepcopy(y), 1, 10, 1) #build decision tree
        ↪  off this
        sf_features.append(len(superfluous)) #append the number of superfluous features, which
        ↪  is just the length of the superfluous dictionary
        #note that since "superfluous" is a dictionary, it stores the unique elements --
        ↪  hence, taking the size is enough to know the number of
        #superfluous features in the tree
```

```
        #we also check if the index is greater than 4 (because that's the definition of a
        ↪ superfluous feature)
        #hence, this length is accurate by the definition of a superfluous feature
        #clear the superflous global variable as we want to start with an empty dictionary
        ↪ each time
        superfluous.clear()

    #Plot/Graph building
    plt.title('"d" value vs. Superfluous features')
    plt.xlabel('d')
    plt.ylabel('Superfluous features')
    plt.plot(indices, sf_features)
    plt.show()
```



The above graph shows the effect that increasing the value of d for the dataset has on the number of superfluous features maintained by the decision tree. We restrict the depth of each decision tree at 10 in order to see the true relationship between the d value and the number of superfluous features. Larger depth values would result in more number of superfluous features being maintained, which would defeat the purpose of this experiment. As per the above graph, we see that as the value of d increases, the number of superfluous features included in the model also linearly increases.

In general, throughout the several experiments we have done, it is clear that truncating by depth is the best way to exclude independent features compared to truncating by sample size. The graph shows that as we increase the number of superfluous features in our dataset, the decision tree thinks that these features carry meaning towards predicting our y value. In reality, we know that these features have no correlation to y. Therefore, in order to restrict the number of superfluous features that our decision tree uses, it is best to reduce the complexity of the model itself. We have discussed this before – the best way to control the complexity of a decision tree is to set a lower maximum depth size. Thus, the best approach to excluding features is to truncate by depth. While sample size may have similar results to truncating by depth, the sample size isn't inherently correlated to the complexity of the model whereas every additional layer within the tree adds a new layer of complexity (as the model identifies a new set of correlations).

Problem 7.

For Linear Regression, the generate_data and compute_mse functions have been slightly adjusted. The following code snippets shows these revisions, the gradient descent algorithm – which has been used to calculate the weights of the linear regression model – and the error from both the training and testing dataset.

```python
import math
import numpy as np

def generate_data_lin_reg(size, d):
    x = []
    x.append(np.ones(size)) #add a column of 1's now to reflect the bias term in a linear regression model
    #the rest stays the same
    x.append(np.random.normal(3, 1, size)) #X_1 as per the problem
    x.append(np.random.normal(-2, 1, size)) #X_2
    x.append(x[1] + 2 * x[2]) #X_3
    x.append((x[2] + 2)**2) #X_4
    x.append(np.random.binomial(n=1, p=0.8, size=size)) #X_5
    for _ in range(d - 5): #if d > 5, then we add d - 5 additional normally distributed random variables
        x.append(np.random.normal(0, 0.1, size))

    def compute_y(x): #use this helper function to compute the y value based on the model in the problem statement
        y = 4 - 3 * x[1] * x[1] + x[3] - 0.01 * x[4] + x[2] * x[5] + np.random.normal(0, 0.1, len(x[0]))
        return y

    def transpose(temp): #a helper method to compute the transpose of the list (also converts to numpy list)
        temp = np.array(temp)
        return temp.T

    xt = transpose(x)
    y = compute_y(x)

    return np.array(x), xt, y

def compute_mse_lin_reg(x, y, w):
    err = 0 #err
    for i in range(len(x)):
        err += abs(np.matmul(w, x[i]) - y[i]) ** 2 #compute the squared error
    err = err / len(x) #mean of the errors
    return err
```

Much of the core of these functions remains the same. The new generate_data function adds a column of ones as we now need to account for the bias term of the linear regression model. The new compute_mse function takes in another parameter, w, which are the weights of the linear regression model.

```python
def gradient_descent(x, xt, y, testx, testy):
    w = [1] * len(xt) #start with an arbitrary set of weights -- I opted for a vector filled with 1s
    xtx = np.matmul(xt, x) #multiply xt and x to get x^t x

    #to find alpha, find a value such that it is greater than 0 but less than 2/(max eigenvalue of xtx)
    #1 / (max eigenvalue of xtx) always follows the above, so I use that each time
    def find_alpha(xtx):
        eigvals = np.linalg.eigvals(xtx)
        mx = np.max(eigvals)
```

```
        return 1 / mx

    alpha = find_alpha(xtx) #store this value
    grad = np.matmul(xtx, w) - np.matmul(xt, y) #find the initial gradient

    #stopping condition is if the gradient is less than 0.1
    #note that in the bonus, I changed this to 2
    #I add this as a comment and didn't make a separate function because the number 2 is
    ↪ somewhat random
    #and I actually changed that because my computer couldn't handle 0.1 for some reason (too
    ↪ many iterations)
    #On a better computer, I think the 0.1 would still hold
    #I discuss more about this on the pdf
    while np.linalg.norm(grad) > 0.1:
        grad = np.matmul(xtx, w) - np.matmul(xt, y)
        w = w - alpha * grad

    train_err = compute_mse_lin_reg(x, y, w) #training_error
    test_err = compute_mse_lin_reg(testx, testy, w) #test_error

    return w, train_err, test_err

xt, x, y = generate_data_lin_reg(10000, 10) #get our linear regression training dataset
_, testx, testy = generate_data_lin_reg(1000, 10) #our testing linear regression dataset
w, train_err, test_err = gradient_descent(x, xt, y, testx, testy) #apply gradient descent

#print weights, training error, and testing error
print('Weights: ')
print(w)
print('Training Error: ', train_err)
print('Testing Error: ', test_err)
```

From the three function calls above, we generate a slightly modified dataset (adding the feature filled with $1's$) to a linear model. We use gradient descent to determine the set of weights that reduces the error of the linear model. The stopping condition for gradient descent is when the norm of the gradient is less than or equal to 0.1, and we found α by taking the reciprocal of the maximum eigenvalue of X^TX. Note that this will always be greater than 0 and $\frac{1}{\lambda_{max}} < \frac{2}{\lambda_{max}}$, making it a viable α for gradient descent. The below screenshot shows the weights of the linear model, the training error, and testing error of the linear model.

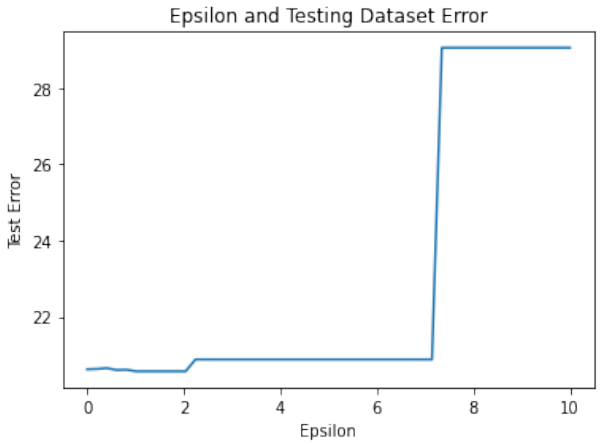
```
Weights:
[ 3.00049855e+01 -1.48683186e+01  7.31245246e+00 -2.24341371e+00
 -4.59213271e-02 -1.98690167e+00 -3.21616922e-01  7.23159987e-02
 -1.26621926e-01  2.67036435e-01 -6.22403605e-04]
Training Error:  17.72121250782168
Testing Error:  18.052209508362218
```

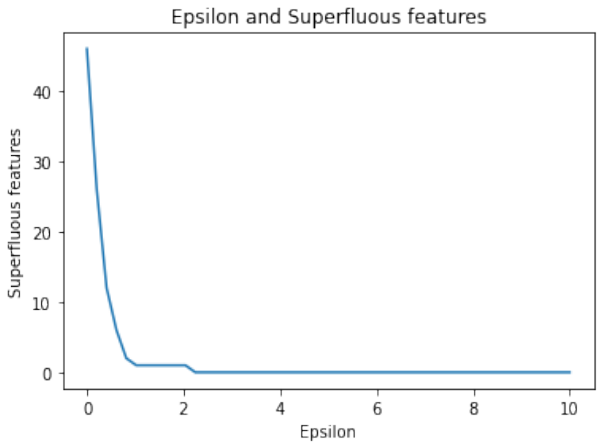
The error of my model on the testing data is substantially smaller than the error of the constant model. Again, this should make sense: a model hasn't learnt anything if it's errors are not substantially lower than merely predicting the mean of the y values each time. Lastly, overfitting should not be a large issue with this model as the training error and testing error are fairly close to each other. In fact, as both errors are very high compared to the decision tree, the linear model may be slightly underfitting to the dataset as a linear model may not be able to capture nonlinear features.

Problem 8.

Solution

```python
xt, x, y = generate_data_lin_reg(10000, 50) #training
_, testx, testy = generate_data_lin_reg(1000, 50) #testing
w, train_err, test_err = gradient_descent(x, xt, y, testx, testy) #apply gradient descent

def question8(xt, y, testx, testy, w):
    espss = np.linspace(0, 10, num = 50) #this will be our array of epsilons that we will conside r
    test_errors = [] #for graph, testing errors and number of superfluous features
    superfluous = [] #superflous features

    testx = testx.T #take transpose of testx matrix

    for eps in espss: #try each epsilon
        wcpy = copy.deepcopy(w) #make a copy of w since we don't want to update w itself
        fltr = abs(wcpy) > eps #filter out the ones that are greater than it and retrain the entire model (using gradient descent) with the new weights

        superfluous.append(fltr[6:].sum()) #we can simply append the number of trues to superfluous
        #note that we slice after the 5th element as beyond that, every feature is a superfluous feature

        #build the new dataset (some numpy tricks to quickly pull all elements from the filter)
        xt_new = xt[fltr]
        testx_new = testx[fltr]
        testx_new = testx_new.T

        x_new = xt_new.T

        #pass it into gradient descent and then retrieve the testing error
        _, _, test_err = gradient_descent(x_new, xt_new, y, testx_new, testy)
        test_errors.append(test_err)

    #Graph building
    plt.title('Epsilon and Testing Dataset Error')
    plt.xlabel('Epsilon')
    plt.ylabel('Test Error')
    plt.plot(espss, test_errors)
    plt.show()

    plt.title('Epsilon and Superfluous features')
    plt.xlabel('Epsilon')
    plt.ylabel('Superfluous features')
    plt.plot(espss, superfluous)
    plt.show()

    return espss, test_errors, superfluous

#function call
epss, _, sf = question8(xt, y, testx, testy, w)
```





Both plots for $d = 50$ are shown above. Evidently, as we see from the testing error graph, the testing error drops till $\epsilon = 2$, and after that, the testing error increases. This behavior is expected. Initially, $\epsilon \leq 2$ removes some of the superfluous features $X_6, \ldots X_{50}$, which is also shown in the second graph, where the superfluous features drops significantly till $\epsilon \leq 2$. Beyond this epsilon, the model drops a small amount of superfluous features while the testing error sharply increases.

Yes, this is a good strategy, so long as ϵ is chosen correctly. Evidently, if you choose a large ϵ (in this case, if you chose $\epsilon > 2$), this strategy is very ineffective, but an epsilon between 1 and 2 allows us to remove some of the superfluous features while reducing the testing error, thus giving us a much more generalized and powerful model.

Problem 9.

Solution The Decision Tree model is superior compared to the constant model, the linear model, and the modified linear model. It should be obvious that the decision tree is better than the constant model – the constant model is simply the average of the y values. Regardless of the value of d, the constant model is around -28 and gives a training/testing error of about 300.

Both the modified linear model and the normal linear model were powerful models that were able to give us reasonable results (errors of about 17 to 18). However, linear models struggle to learn nonlinearity (unless the dataset is transformed accordingly) and through noise, which showed as the error of the linear model was significantly higher than the decision tree.

The decision tree had a testing error of approximately 0.5 (across $d = 10, 50$), indicating a significant improvement from the linear model. The decision tree was able to learn the nonlinearity of the dataset even through the superfluous features and give us fairly accurate results. Hence, it is definitely the superior model out of the three.

Bonus Question

Below is the code for this question. Note that I did not show my generate_data functions as I didn't want to take up extra space (the functions are very similar; the only difference is that I have a nested for loop that adds the quadratic terms).

```
def questionBonus():
    def compute_mse_constant_model(y, c):
        err = 0
        for i in range(len(y)):
            err += (c - y[i]) ** 2
        err = err / len(y) #take average of squared sum
        return err

    xt, x, y = generate_data_lin_reg_bonus(10000)
    _, testx, testy = generate_data_lin_reg_bonus(1000)

    optimal_c = np.mean(y)
    #Find the errors of both training and testing
    training_error = compute_mse_constant_model(y, optimal_c)
    testing_error = compute_mse_constant_model(testy, optimal_c)

    #Constant Model
    print('Training Error:', training_error)
    print('Testing Error: ', testing_error)

    w, train_err, test_err = gradient_descent(x, xt, y, testx, testy)

    print('Weights: ')
    print(w)
    print('Training Error: ', train_err)
    print('Testing Error: ', test_err)

    x, xt, y = generate_data_bonus(10000)
    _, tx, testy = generate_data_bonus(1000)
    dt = DecisionTree(copy.deepcopy(x), copy.deepcopy(y), 1, 15, 5)

    training_err = compute_mse_dt(xt, y, dt)
    testing_err = compute_mse_dt(tx, testy, dt)

    print('Training Error:', training_err)
    print('Testing Error: ', testing_err)
```

```
Weights:
[ 1.78028401e+00+0.j  1.23798529e+00+0.j  2.62207953e-01+0.j
 -2.37598808e-01+0.j  5.42890764e-01+0.j  1.45363413e-03+0.j
  2.29070386e-01+0.j -2.21821256e+00+0.j  2.02709811e+00+0.j
 -1.64016332e-01+0.j  3.33317758e-01+0.j -8.36310762e-04+0.j
  2.93725756e-01+0.j  3.72922920e-01+0.j  7.72943954e-01+0.j
  6.66621436e-01+0.j  1.00045273e+00+0.j  6.90165093e-01+0.j
 -6.18128424e-01+0.j -3.33439369e-01+0.j  6.91566392e-05+0.j
 -3.25944058e-01+0.j -1.97635925e-04+0.j  1.23972541e-03+0.j
 -1.87678993e-02+0.j  1.45363413e-03+0.j -1.84857784e-02+0.j
  7.00620923e-01+0.j]
Training Error:  0.010254352819320679
Testing Error:  0.010194766009707605
```

Reviewing the results, the constant model had no difference. The training error was around 317.933 and the testing error was roughly 326.13. This should be of no surprise; as explained before, the constant model does not change as long as the y function does not change. The training/testing errors and the weights of the linear regression model are shown above. The training and testing error of the decision tree was 0.193 and 0.3046, respectively.

Evidently, the linear regression model was the best performing model. However, by adding the quadratic terms, in a sense, we forcibly fit the linear regression model to the dataset. This leads to an overfitting of the linear regression model to this dataset and a loss of generality. For this particular scenario, where we know the function we are predicting beforehand, this is fine as we are catering to the relationship between the x and y values. In practical applications, we will never know the function that we are predicting beforehand. Thus, the decision tree model is still a powerful model and is potentially more versatile than the overfit linear regression model.