CPT_S 570

Machine Learning, Fall 2020

Homework # 3

Tayyab Munir

11716089

Analytical Section

Question 1:

Part a.

$f(Y) = Y^2$

$X - Z = Y$

$f(E[Y]) \leq E[f(Y)]$

$(E[X - Z])^2 \leq E[(X - Z)^2]$

$$\left(\frac{(x_i - z_i) + \cdots + (x_n - z_n)}{d}\right)^2 \leq \frac{(x_i - z_i)^2 + \cdots + (x_n - z_n)^2}{d}$$

$$\left(\frac{\Sigma x_i - \Sigma z_i}{\sqrt{d}}\right)^2 \leq \Sigma \frac{(x_i - z_i)^2}{d}$$

Part b.

If d is very large then in order to compute the actual dot product a dot product of very large dimension has to be compute and By using Jensen inequality based approximation approximate distance can be computed without a very large order dot product. Approximate nearest neighbor algorithm can be devised based on this inequality.

Question 2:

Authors focus on methods to find approximate nearest neighbors in high dimensions when finding the exact nearest neighbor could be computationally very difficult and time consuming given the huge search space. Authors also classify the nearest neighbor problem into two classes: First class is finding several near neighbors and iteratively finding approximate nearest neighbor. Second class is finding nearest neighbor. Locality sensitive hashing is an approximate near neighbor search algorithm. Locality sensitive hashing algorithm relies on the existence of locality sensitive hash functions. Hashing functions are randomly picked from family of hashing functions and probability of h(p)=h(q) is analyzed. Authors describe the usefulness of Locality sensitive family in terms of probabilities. Gap between probabilities is magnified by concatenating several hash functions. Hash functions are picked randomly from family” H. A point “q” is scanned through buckets to retrieve points from hashing buckets. Two scanning strategies are considered i.e. stopping when L’ points have been retrieved or all buckets have been scanned.

Question 3:

It would be possible to convert a rule dictionary to decision tree only if we can figure the order of rules. Some algorithms can be devised to look at the rules and decide which rule for instance provides most information gain and that could be the root node. As we follow the rules something similar can be done to choose order of child nodes.

Question 4:

Authors present theoretical and empirical results-based comparison of generative and discriminative classifiers. Authors also pose that although the general belief is that discriminative classifiers perform better but there can be two different regimes. In first regime, generative model approaches the asymptotic error faster and hence performs better than their generative counterparts. In second regime discriminative classifiers converge on asymptotic error faster and their performance is better. Asymptotic error of a classifier is the error as training examples approach infinity. Authors propose that Asymptotic error of discriminative classifiers is less than generative classifiers. Moreover, the authors proposed and experimentally investigated the idea that as the number of training examples are increased generative classifier would initially do better because of faster convergence to asymptotic error but ultimately discriminative classifier overtakes it and performs better because of low asymptotic error.

Question 5:

Part a.

Under conditional independence assumption both classifiers would produce equivalent results when number of training examples approach infinity.

Part b.

In this case since features are not conditionally independent logistic regression should perform better since it does not assume conditional independence in problem formulation.

Part c.

We can compute P(X) because in problem formulation Naive bayes classifier learns $P(\frac{X}{Y})$. e-g for a binary classifier $P(X = X_k) = P(X = X_k).P(Y = 0) + P(X = X_k).P(Y = 1)$. All these parameters are learnt in Naive bayes classifier learning so P(X) can be computed.

Part d.

Logistic regression is a discriminative classifier and since it does not learn $P(\frac{X}{Y})$ it cannot compute $P(X)$.

Question 6:

In this question author has discussed over-fitting and under-computing in context of Machine learning algorithms. Under-computing helps us to avoid over-fitting the training data. Authors point towards the fact that the goal of machine learning algorithms is to generalize better i.e., do well on new data points. Just trying to do best possible job on training data might make machine learning algorithm memorize peculiarities of training data rather than learning important generalization details needed to do better on new data points.

Author also points toward different methodologies to deal with over-fitting problem like cross validation regularization etc. which help the machine learning algorithms to generalize better. Author also points that greedy sub-optimal algorithms like gradient descent and greedy method to find decision tree also to generalize better because they can naturally deal with over-fitting by under-computing.

Question 7:

Paper reviews 5 statistical tests for determining the better supervised learning algorithm for task among a set of algorithms. Probability of incorrectly detecting a difference when no difference exists is called Type-I Error. Authors compare the five tests with respect to their vulnerability to type I error and power of tests (i.e. ability to identify Algorithmic differences) Authors propose a new statistical test and prove its superiority through experimentation. Authors present 9 statistical questions. Which mainly concern the sample size and testing and comparing the classifier algorithms for various scenarios.

In Paper, Question 8: Given a small Sample set S, which of the considered classification algorithm will perform best if trained on all data. Authors consider 5 statistical tests to answer this question. McNemar’s Test: Data is divided into training and testing set. The classifiers are then tested on test set for comparison. It is based on Chi-squared test. Slightly less powerful than 5x2cv but is computationally efficient hence preferred for expensive learning methods. They compute a test value which checks if null hypothesis (Both classifiers have same performance) holds or not. Authors conclude that this is best possible option for such learning algorithms. One of the shortcomings is that this test does not consider variability due to choose of training data. If sources of variability are small this test can be applied confidently. A test for the difference of two proportions: Authors note that This test performs badly in some cases but is computationally very cheap. It is based on measuring error rate of compared classification algorithms. The problem with this test is that proportions pa and pb are measured on same training set R. This test shares the drawbacks of McNbemar’s test. The resampled T-test: Type I error is found to be very bad for this test. This test is also computationally expansive. The k-cross Validated T-test: Sample set S is divided into k disjoint sets and for every trial a different set is chosen for test and rest for Training. This test still suffers from the problem that training sets overlap. 5x2cv Test: Powerful but also quite expensive to perform. In this test authors perform 5 replications of 2-fold cv test. In each replication data is partitioned into 2 sets: Training and Testing. Based on 5 such replications and respective testing t-statistic is computed. After describing the tests authors compare the simulation behavior. Simulation results show that only McNemars, the cross validated t-test and 5x2cv test are good with respect to type I error.

Programming section

Question 1:

[1]:
```
import pandas as pd
import numpy as np
from scipy import linalg, optimize
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import random
from sklearn import svm
```

1 Loading Data

[2]:
```
file = open('./data/traindata.txt', 'r')
FC_Lines_Train = file.readlines()
FC_Lines_Train=[x.strip() for x in FC_Lines_Train]
file.close()
file = open('./data/stoplist.txt', 'r')
Stop=file.readlines()
Stop = [x.strip() for x in Stop]
file.close()
file = open('./data/testdata.txt', 'r')
FC_Lines_Test = file.readlines()
FC_Lines_Test=[x.strip() for x in FC_Lines_Test]
file.close()
file = open('./data/testlabels.txt', 'r')
y_test = [[int(v) for v in line.split()] for line in file]
y_test=np.array(y_test)
file.close()
y_test=np.array(y_test)
file = open('./data/trainlabels.txt', 'r')
y_train = [[int(v) for v in line.split()] for line in file]
y_train=np.array(y_train)
file.close()
y_train=np.array(y_train)
```

2 Generating Vocabulary and Features

[3]:
```
def Generate_Vocabulary(FC_Lines,Stop):
    W=[]
    for line in FC_Lines:
        W.extend(line.strip().split(' '))
    W=sorted(W)
    Vocabulary=[]
    for i in range(len(W)):
        if W[i] not in Stop:
            if W[i] not in Vocabulary:
                Vocabulary.append(W[i])
    Count=[]
    for i in range(len(Vocabulary)):
        CNT=0
        for j in range(len(W)):
            if(Vocabulary[i]==W[j]):
                CNT=CNT+1
        Count.append(CNT)
    return Vocabulary,Count
def Generate_Features(FC_Lines,Vocabulary):
    X_train=np.zeros([len(FC_Lines),len(Vocabulary)])
    for i in range(len(FC_Lines)):
        Temp=FC_Lines[i].strip().split(' ')
        for k in range(len(Vocabulary)):
            if Vocabulary[k] in Temp:
                X_train[i,k]=1
    return X_train
```

[4]:
```
[Vocabulary,Count]=Generate_Vocabulary(FC_Lines_Train,Stop)
```

[5]:
```
X_train=Generate_Features(FC_Lines_Train,Vocabulary)
X_test=Generate_Features(FC_Lines_Test,Vocabulary)
```

3 Training Naive Bayes Classifier: Calculating Naive Bayes Probabilities

[6]:
```
def Probabilities(y_train,X_train):
    Count_Yes=0
    Count_Pos_1=np.zeros([len(X_train[0]),1])
    Count_Pos_0=np.zeros([len(X_train[0]),1])
    for i in range(len(y_train)):
        if((y_train[i][0])==1):
            Count_Yes+=1
            for k in range(len(X_train[i])):
                if((X_train[i][k])==1):
                    Count_Pos_1[k]+=1
        else:
            for k in range(len(X_train[i])):
                if((X_train[i][k])==1):
                    Count_Pos_0[k]+=1
    Pr_Yes=Count_Yes/len(y_train)
    Pr_No=1-Pr_Yes
    Count_No=len(y_train)-Count_Yes
    Pos_prob=(Count_Pos_1+1)/(Count_Yes+2)
    Neg_prob=(Count_Pos_0+1)/(Count_No+2)
    CP=(Count_Pos_1+Count_Pos_0)
    PXi=(CP+1)/(len(y_train)+2)
    return Pr_Yes,Pr_No,Pos_prob,Neg_prob,PXi
```

[7]:
```
[Pr_Yes,Pr_No,Pos_prob,Neg_prob,PXi]=Probabilities(y_train,X_train)
```

4 Predict Function

[8]:
```
def predict(X_testk,Pr_Yes,Pr_No,Pos_prob,Neg_prob,PXi):
    Pr_XP=Pr_Yes
    Pr_XN=Pr_No
    PX=1
    for i in range(len(X_testk)):
        if((X_testk[i])==1):
            Pr_XP=Pr_XP*Pos_prob[i]
            Pr_XN=Pr_XN*Neg_prob[i]
            PX=PX*PXi[i]
        else:
            Pr_XP=Pr_XP*(1-Pos_prob[i])
            Pr_XN=Pr_XN*(1-Neg_prob[i])
            PX=PX*(1-PXi[i])
    P_1=Pr_XP/PX
    P_0=Pr_XN/PX
    if(P_1>=P_0):
        y_pred=1
    else:
        y_pred=0
    return y_pred
```

5 Accuracy for Training and Testing Data

[9]:
```
Rights=0
for i in range(len(y_test)):
    ypred=predict(X_test[i],Pr_Yes,Pr_No,Pos_prob,Neg_prob,PXi)
    if (ypred==y_test[i][0]):
        Rights=Rights+1
Testing_Accuracy=Rights/len(y_test)
Rights=0
for i in range(len(y_train)):
    ypred=predict(X_train[i],Pr_Yes,Pr_No,Pos_prob,Neg_prob,PXi)
    if (ypred==y_train[i][0]):
        Rights=Rights+1
Training_Accuracy=Rights/len(y_train)
print('Test Accuracy:',Testing_Accuracy,'\nTrain Accuracy:',Training_Accuracy)
```

```
Test Accuracy: 0.7722772277227723
Train Accuracy: 0.9316770186335404
```

Question 2:

1 Deep Learning

[2]:
```python
import matplotlib.pyplot as plt
from torch import tensor
import torch
import matplotlib as mpl
import pandas as pd
import torch.nn as nn
import torch.nn.functional as F
from torch import optim
```

1.1 Loading dataset

[3]:
```python
def get_data():
    train_data = pd.read_csv('./data/fashion-mnist_train.csv')
    test_data = pd.read_csv('./data/fashion-mnist_test.csv')
    x_train = train_data[train_data.columns[1:]].values
    y_train = train_data.label.values
    x_test = test_data[test_data.columns[1:]].values
    y_test = test_data.label.values
    return map(tensor, (x_train, y_train, x_test, y_test)) # maps are useful␣
 ↪functions to know
                                                            # here, we are just␣
 ↪converting lists to pytorch tensors
```

[5]:
```python
x_train, y_train, x_test, y_test = get_data()
train_n, train_m = x_train.shape
test_n, test_m = x_test.shape
n_cls = y_train.max()+1
```

[24]:
```python
y_train=y_train[0:10]
x_train=x_train[0:10]
```

[6]:
```python
train_n, train_m = x_train.shape
test_n, test_m = x_test.shape
n_cls = y_train.max()+1
```

1.2 Creating a model

Convolutional Neural Network Model is created in this part: * First Convolutional Layer has 8 Outputs, stride length of 2, and Filter size of 5x5 is used. This produces 8 outputs of 14x14 convolutional layers. Convolutional layers are followed by RELU layer. * Second covolutional layer has 16 outputs each a 7x7 convolutional layer. Convolutional layers are followed by RELU layer. * Third Convolutional layer: This produces 32 outputs each a 4x4 convolutional layer. Convolutional layers are followed by RELU layer. * Fourth Convolutional layer: This produces 32 outputs each a 2x2 convolutional layer. Convolutional layers are followed by RELU layer. * Polling Layer: Polling layer produces 32 outputs each is 1x1 scalar. * Fully connected layer takes 32 1x1 inputs and 10 class labels * A relu activation function is also used after pooling layer to enhance accuracy.

[39]:
```
# Definition of the model
class FashionMnistNet(nn.Module):
    # Based on Lecunn's Lenet architecture
    def __init__(self):
        super(FashionMnistNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 5,2,2)
        self.conv2 = nn.Conv2d(8, 16, 3,2,1)
        self.conv3 = nn.Conv2d(16, 32, 3,2,1)
        self.conv4 = nn.Conv2d(32, 32, 3,2,1)
        self.mp=nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(32, 10)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        #print(x.size())
        x = F.relu(self.conv2(x))
        #print(x.size())
        x = F.relu(self.conv3(x))
        #print(x.size())
        x=self.conv4(x)
        #print(x.size())
        x = F.relu(self.mp(x))
        #print(x.size())
        x = x.view(-1, self.num_flat_features(x))
        #print(x.size())
        x = self.fc(x)
        return x
    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

1.3 Training the model

[40]:
```
model = FashionMnistNet()
print(model)
model.forward(x_train[0].reshape(1, 1, 28, 28))
```

```
FashionMnistNet(
  (conv1): Conv2d(1, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv4): Conv2d(32, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (mp): AdaptiveAvgPool2d(output_size=1)
  (fc): Linear(in_features=32, out_features=10, bias=True)
)
torch.Size([1, 8, 14, 14])
torch.Size([1, 16, 7, 7])
torch.Size([1, 32, 4, 4])
torch.Size([1, 32, 2, 2])
torch.Size([1, 32, 1, 1])
torch.Size([1, 32])
```

[40]:
```
tensor([[-0.0236,  0.1181, -0.0468, -0.1567, -0.1847, -0.1630,  0.1054,  0.1321,
          0.0940, -0.0844]], grad_fn=<AddmmBackward>)
```

[31]:
```
### Normalization
x_train, x_test = x_train.float(), x_test.float()
train_mean,train_std = x_train.mean(),x_train.std()
train_mean,train_std
```

[31]:
```
(tensor(1.3403e-05), tensor(1.))
```

[32]:
```
def normalize(x, m, s): return (x-m)/s
x_train = normalize(x_train, train_mean, train_std)
x_test = normalize(x_test, train_mean, train_std) # note this normalize test␣
 ↪data also with training mean and standard deviation
```

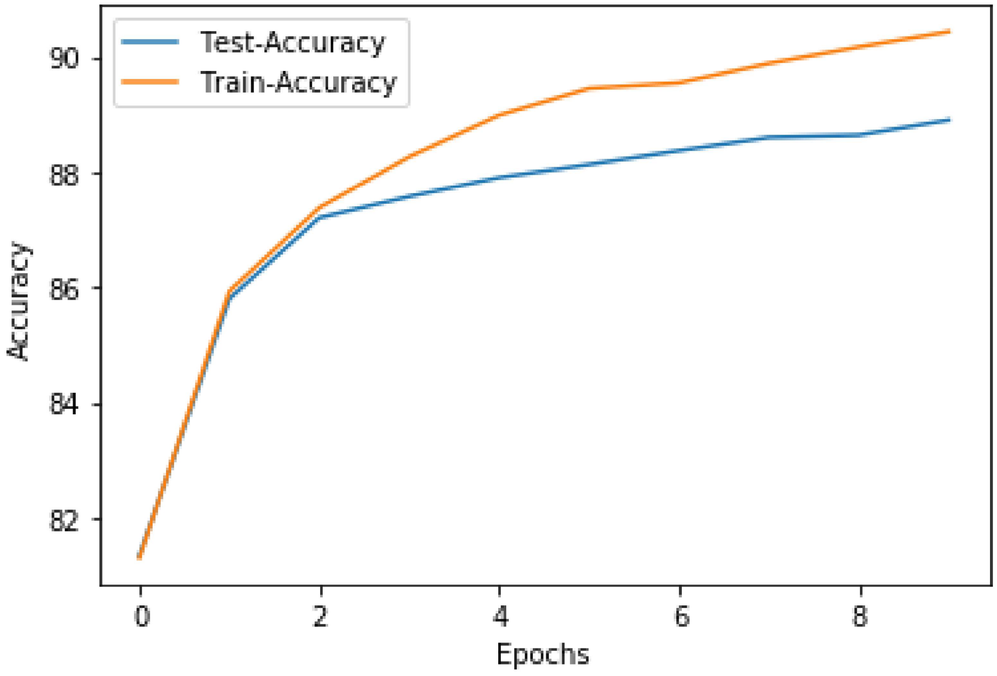
[34]:
```
model_wnd = FashionMnistNet()
lr = 0.1 # learning rate
epochs = 10 # number of epochs
bs = 32
loss_func = F.cross_entropy
opt = optim.ASGD(model_wnd.parameters(), lr=lr)
accuracy_vals_wnd = []
train_accuracy_vals_wnd = []
for epoch in range(epochs):
    model_wnd.train()
    for i in range((train_n-1)//bs + 1):
        random_idxs = torch.randperm(train_n)
        start_i = i*bs
        end_i = start_i+bs
        xb = x_train[start_i:end_i].reshape(bs, 1, 28, 28)
        yb = y_train[start_i:end_i]
        loss = loss_func(model_wnd.forward(xb), yb)
        loss.backward()
        opt.step()
        opt.zero_grad()

    model_wnd.eval()
    with torch.no_grad():
        total_loss, accuracy = 0., 0.
        validation_size = int(test_n/10)
        for i in range(test_n):
            x = x_test[i].reshape(1, 1, 28, 28)
            y = y_test[i]
            pred = model_wnd.forward(x)
            accuracy += (torch.argmax(pred) == y).float()
        print("Accuracy_Test: ", (accuracy*100/test_n).item(),test_n,accuracy)
        accuracy_vals_wnd.append((accuracy*100/test_n).item())
    model_wnd.eval()
    with torch.no_grad():
        total_loss_train, accuracy_train = 0., 0.
        validation_size = int(train_n/10)
        for i in range(train_n):
            x = x_train[i].reshape(1, 1, 28, 28)
            y = y_train[i]
            pred = model_wnd.forward(x)
            accuracy_train += (torch.argmax(pred) == y).float()
        print("Accuracy_Train: ", (accuracy_train*100/train_n).
 ↪item(),test_n,accuracy_train)
        train_accuracy_vals_wnd.append((accuracy_train*100/train_n).item())
```

```
Accuracy_Test:  81.33999633789062 10000 tensor(8134.)
Accuracy_Train:  81.30333709716797 10000 tensor(48782.)
Accuracy_Test:  85.80000305175781 10000 tensor(8580.)
Accuracy_Train:  85.93000030517578 10000 tensor(51558.)
Accuracy_Test:  87.19999694824219 10000 tensor(8720.)
Accuracy_Train:  87.37333679199219 10000 tensor(52424.)
Accuracy_Test:  87.56999969482422 10000 tensor(8757.)
Accuracy_Train:  88.26333618164062 10000 tensor(52958.)
Accuracy_Test:  87.9000015258789 10000 tensor(8790.)
Accuracy_Train:  88.98500061035156 10000 tensor(53391.)
Accuracy_Test:  88.12999725341797 10000 tensor(8813.)
Accuracy_Train:  89.44999694824219 10000 tensor(53670.)
Accuracy_Test:  88.37000274658203 10000 tensor(8837.)
Accuracy_Train:  89.54499816894531 10000 tensor(53727.)
Accuracy_Test:  88.5999984741211 10000 tensor(8860.)
Accuracy_Train:  89.88333129882812 10000 tensor(53930.)
```

```
Accuracy_Test:  88.63999938964844 10000 tensor(8864.)
Accuracy_Train:  90.1683349609375 10000 tensor(54101.)
Accuracy_Test:  88.9000015258789 10000 tensor(8890.)
Accuracy_Train:  90.4316635131836 10000 tensor(54259.)
```

[38]:
```
P1=plt.plot(accuracy_vals_wnd)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
P2=plt.plot(train_accuracy_vals_wnd)
plt.legend(('Test-Accuracy','Train-Accuracy'))
```

[38]:
```
<matplotlib.legend.Legend at 0x22b80330f98>
```


Test-Accuracy
Train-Accuracy
Accuracy
Epochs