CS 470 Final Project Implementation and Performance

Email Spam Classification using Naïve Bayes, Linear Regression, and K Nearest Neighbor

GitHub: https://github.com/Devin-Jay/EmailSpamClassification

Project Details

1. Import Dataset.

```
def importDataset(fileName):
    with open(fileName) as file:
        dataset = file.readlines()
        for email in range(len(dataset)):
            dataset[email] = dataset[email].split(",")
    return dataset
```

2. Get training set and evaluation set.

```
# get dataset
dataset = importDataset("spambase.csv")

# initialize model classes with data labels
NBAlgorithm = NaiveBayesModel(dataset[0])
LRClass = LogisticRegression(dataset[0])
KNNClass = KNearestNeighbor(dataset[0])

#shuffle dataset
dataset = dataset[1:]
random.shuffle(dataset)

# get training set
trainingSize = math.floor(trainingPct * (len(dataset)))
trainingSet = dataset[:trainingSize + 1]
random.shuffle(trainingSet)

# get evalSet and evalSet actualvals
evalSet = dataset[trainingSize:]
evalActualResults = [int(email[-1]) for email in evalSet]
```

3. Divide the training set into K fold, which in this case is 5.

I did this by looping through the training set and manipulating the indices to get the fold

```
# loop through training (increment by size of training // 5 for 5 cross validation)
for fold in range(0, len(trainingSet) - (len(trainingSet) // K), len(trainingSet) // K):
```

4. For each fold:
    a. Train model where applicable.

```
# train Naive Bayes model and test on fold
predictedResults, spamPrior, hamPrior, spamLikelihoods, hamLikelihoods = NBAlgorithm.naiveBayes(fold, fold + len(trainingSet) // K, trainingSet)
```

```
# train Linear Regression model and test on fold
predictedResults, model = LRClass.LRAlgorithm(fold, fold + len(trainingSet) // K, trainingSet)
```

b. Evaluate performance on fold.

```
# get actualResults
actualResults = [int(email[-1]) for email in trainingSet[fold:fold + len(trainingSet) // K]]

# get performance results
acc, fp, tp, auc = perf.performance(predictedResults, actualResults)

# add to fold results
foldResults.append((acc, fp, tp, auc))
```

c. Use model on evaluation set.

```
# test on evalSet
result = NBAlgorithm.useModel(spamPrior, hamPrior, spamLikelihoods, hamLikelihoods, evalSet)
```

```
# test on evalSet
result = LRClass.useModel(model, evalSet)
```

```
# test on evalSet
result = KNNClass.classifySet(evalSet, compareEmails)
```

5. Display results.

```
Naive Bayes Fold Average:
Accuracy:  0.8894021739130433
False Positives:  0.7723297969486673
True Positives:  0.41776958357960253
Area Under Curve:  0.8974213044661334

Naive Bayes EvalSet Average:
Accuracy:  0.9064060803474483
False Positives:  0.6820563362765826
True Positives:  0.3900344807435624
Area Under Curve:  0.9094292383194349

Linear Regression Fold Average::
Accuracy:  0.6921195652173914
False Positives:  0.5599136461350878
True Positives:  0.37781303186553916
Area Under Curve:  0.6868973673980285

Linear Regression EvalSet Average::
Accuracy:  0.71357220412595
False Positives:  0.499903715257473
True Positives:  0.33664934052614004
Area Under Curve:  0.6971172644934764

K Nearest Neighbor Fold Average::
Accuracy:  0.8111413043478262
False Positives:  0.46992923751359256
True Positives:  0.36605876939613713
Area Under Curve:  0.8004256033455966

K Nearest Neighbor EvalSet Average::
Accuracy:  0.8108577633007601
False Positives:  0.528002440331042
True Positives:  0.36261445769162953
Area Under Curve:  0.8025834896061923
```

Project Classes

Naïve Bayes Model Class

1. Get training set
2. Train model by calculating attribute likelihoods and spam and non-spam probabilities
3. Use model on eval set

```
def naiveBayes(self, evalStartIndex, evalEndIndex, dataset):
    # initialize training set (all other folds, not current fold)
    trainingSet = dataset[:evalStartIndex] + dataset[evalEndIndex + 1:len(dataset)]

    # get likelihoods and priors
    spamPrior, hamPrior, spamLikelihoods, hamLikelihoods = self.calculateLikelihoods(trainingSet)

    # get dataset to be tested on (current fold)
    evalSet = dataset[evalStartIndex:evalEndIndex]

    # test current model on current fold
    result = self.useModel(spamPrior, hamPrior, spamLikelihoods, hamLikelihoods, evalSet)

    return result, spamPrior, hamPrior, spamLikelihoods, hamLikelihoods
```

```
def calculateLikelihoods(self, dataset):
    # get data that was classified as spam
    spamData = self.getClassifiedData(dataset, True)
    spamDataSize = len(spamData)

    # get data that was classified as normal
    hamData = self.getClassifiedData(dataset, False)
    hamDataSize = len(hamData)

    # initialize counts with 1 (laplace smoothing)
    spamLikelihoods = [1] * (len(self.labels) - 4)
    hamLikelihoods = [1] * (len(self.labels) - 4)

    totalSpamWords = len(self.labels) - 4
    totalHamWords = len(self.labels) - 4

    for attribute in range(len(self.labels) - 4):
        for email in range(1, spamDataSize):
            if (float(spamData[email][attribute]) > 0.0):
                spamLikelihoods[attribute] += 1
                totalSpamWords += 1

        for email in range(1, hamDataSize):
            if (float(hamData[email][attribute]) > 0.0):
                hamLikelihoods[attribute] += 1
                totalHamWords += 1

    for attribute in range(len(self.labels) - 4):
        spamLikelihoods[attribute] /= totalSpamWords
        hamLikelihoods[attribute] /= totalHamWords

    spamPrior = spamDataSize / (spamDataSize + hamDataSize)
    hamPrior = hamDataSize / (spamDataSize + hamDataSize)

    return spamPrior, hamPrior, spamLikelihoods, hamLikelihoods
```

```python
# function that returns list of emails that are classified as yes or no (depends on parameter input)
def getClassifiedData(self, dataset, spam):
    classifiedData = []

    for email in range(len(dataset)):
        if (int(dataset[email][-1]) == spam):
            classifiedData.append(dataset[email])

    return classifiedData
```

```python
def useModel(self, spamPrior, hamPrior, spamLikelihoods, hamLikelihoods, evalSet):
    emailPredictions = []

    for email in range(len(evalSet)):
        emailsSpamLikelihoods = 1
        emailsHamLikelihoods = 1

        for attribute in range(len(self.labels) - 4):
            if (float(evalSet[email][attribute]) > 0.0):
                emailsSpamLikelihoods *= spamLikelihoods[attribute]
                emailsHamLikelihoods *= hamLikelihoods[attribute]

        emailsSpamLikelihoods *= spamPrior
        emailsHamLikelihoods *= hamPrior

        if (emailsSpamLikelihoods > emailsHamLikelihoods):
            emailPredictions.append(1)
        else:
            emailPredictions.append(0)

    return emailPredictions
```

Logistic Regression Class

1. Get training set
2. Train model by fitting a line to training set
3. Use model on eval set

```python
def LRAlgorithm(self, startIndex, endIndex, dataset):
    # get training set
    trainingSet = dataset[:startIndex] + dataset[endIndex + 1:len(dataset)]

    # fit logistic regression line to graph
    weights = self.gradientDescent(trainingSet, 0.000001, 1000)

    # test on fold
    results = self.useModel(weights, dataset[startIndex:endIndex])

    return results, weights
```

```python
def gradientDescent(self, trainingSet, learningRate, iterations):
    # get classifiers of all emails in trainingSet
    yVal = np.array(self.getYVal(trainingSet))

    # get trainingSet without target column
    xWithBias = []
    for email in trainingSet:
        floatEmail =  [round(float(x),2) for x in email]
        xWithBias.append([0] + floatEmail)
    xWithBias = np.array(xWithBias)

    # get len of dataset
    m, n = xWithBias.shape

    # get model initialized with 0s
    model =  np.zeros(n)

    # fit line
    for i in range(iterations):
        predY = self.sigmoid2(np.dot(xWithBias,model))
        gm = np.dot(xWithBias.T, (predY - yVal)) / m
        model -= (learningRate * gm)

    return model
```

```python
def getYVal(self, trainingSet):
    yVal = []
    for email in range(len(trainingSet)):
        yVal.append(int(trainingSet[email][-1]))

    return yVal

def sigmoid2(self, x):
    clippedX = np.clip(x,-500,500)
    return 1 / (1 + np.exp(-clippedX))
```

```python
def useModel(self, weights, evalSet):
    emailPredictions = []

    for email in evalSet:
        prediction = 0
        for attribute in range(1,len(email) - 1):
            prediction += weights[attribute] * float(email[attribute])

        prediction += weights[0]

        if (prediction > 0.5):
            emailPredictions.append(1)
        else:
            emailPredictions.append(0)

    return emailPredictions
```

K Nearest Neighbor Class

1. Get training set without target classifier
2. Get eval set
3. Calculate the Euclidian distances between each email in eval set and training set
4. Classify each email in eval set using class of email with least Euclidian distance

```python
def classifySet(self, evalSet, trainingSet):
    # get trainingSet without target column
    trainingEmails = [row[:-1] for row in trainingSet]

    # get evalSet without target column
    if (len(evalSet[0]) == len(trainingSet[0])):
        evalEmails = [row[:-1] for row in evalSet]

    # calculate distance between each email in evalSet and trainingEmails
    distances = pairwise_distances(evalEmails, trainingEmails)

    # get prediction for each email
    result = []
    for email in range(len(evalSet)):
        # get class of nearest neighbor to current email
        result.append(int(trainingSet[np.argmin(distances[email])][-1]))

    return result
```