Project 1: Prototype Selection for Nearest Neighbor

Jeff Makings

1 High-level Description

In this prototype selection algorithm, the strategy was to select M points at random from the MNIST training set, build a classifier with these points, and test their performance on a randomly selected validation set also taken from the MNIST training set.

With this performance data, the training data points that lead to correct predictions on the validation set are kept as training data for another round, and the training data that was not used or lead to incorrect predictions is replaced by new, randomly selected training data. Then, this new training set is used to build another mini-classifier, and the process repeats.

This process of keeping ”good” training data, replacing the other data, then testing on new, random validation sets is repeated for 50 iterations, and finally the training data is returned as the prototypes to build a classifier for the test set.

With this unique approach to identifying prototypes, the classification accuracy for 1-Nearest Neighbor was improved for most values of M, with certain drawbacks.

2 Pseudocode

Input: Labeled MNIST training set (MNISTtrain) and the integer M
Output: Subset of MNISTtrain of size M

1.Get M initial training data points from MNIST training set

```
$trainSet$ ← random.sample(MNISTtrain, $M$)
```

2 Pseudocode (continued)

2.Build classifier, test on random validation set of size 10,000

```
NNclassifier.fit($trainSet$)
int $s$ ← 10,000
$validSet$ ← random.sample(MNISTtrain,$s$)
$predictions$ ← NNclassifier.test($validSet$)
```

3.Get indices of correct predictions by comparing to validation set, then get correctly predicted X validation points from these indices

```
Array $correctIndex$[]
For $p$ in range(predictions):
    If predictions[$p$] == validSet[$p$]:
        $correctIndex$.append($p$)

Array $correctX$[]
For $c$ in $correctIndex$:
    $correctX$.append(validSetX[$c$])
```

4.From this array of correctly predicted validation set X values, use the Nearest Neighbor classifier's $kneighbors$ function to get an array of nearest neighbors from the training set to $correctX$'s points

```
$neighbors$ ← NNclassifier.kneighbors($correctX$)
```

3 Experimental Methods and Results

MNIST data was obtained via the Tensorflow module Keras.datasets.mnist. The MNIST data from Tensorflow was already split into 60,000 training and 10,000 test samples. The training set were used immediately for prototype selection and the testing set was set aside.
The algorithm for prototype selection was

2 Pseudocode (continued)

5.This function takes as input the index of the training set point and returns the X and Y training set values at this point. From this, we get ”good” prototypes from the indices

```
Array $goodProto[]$
For $n$ in $neighbors$:
    $point \leftarrow trainSet[n]$
    $goodProto$.append($point$)
```

6.We now have an array of the training set points which were successful nearest neighbors in predicting validation set points. Now we must append randomly selected training data of size ($M-$ len($goodProto[]$)) to the $goodProto$ array to create a new training set of size M.

```
int $s \leftarrow (M$ - len($goodProto$) )
$newSamples \leftarrow$ random.sample(MNISTtrain,$s$)
Array $newTrainData[]$
$newTrainData \leftarrow goodProto + newSamples$
```

7.$newTrainData$ now has the old, ”good” prototypes and new random samples together. The algorithm now returns to Step 2 and $newTrainingData$ is used to train a new classifier. After 50 iterations, the training set is returned as the prototypes for the test set.

evaluated by using selected prototypes to train 1-Nearest Neighbor classifiers via the module Sklearn.neighbors.KNeighborsClassifier and setting $K = 1$. The performance of each classifier was then evaluated on the entire MNIST test set, and the classifier accuracy score was recorded.

Because the algorithm involved some random selection of prototypes,the prototype selection algorithm was tested 10 times for each value of M. For control samples, classifiers trained by 10 uniform random selections of prototypes were also evaluated. These prototypes were obtained by using the library function random.sample to sample M prototypes from the MNIST training data.

With classifier performance data collected, the standard deviation is computed for each set of samples via the formula:

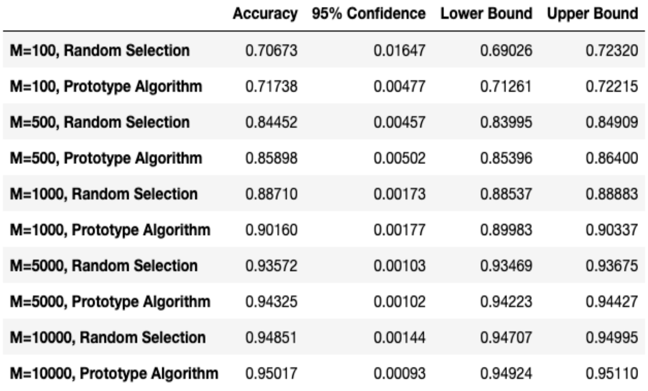
$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

N = number of samples (10)
x_i = observed classifier accuracy for sample i
$\bar{x}$ = mean classifier accuracy for set

Using the standard deviation for each set of classifiers, the 95% confidence interval can be computed via the formula:

$$CI = \bar{x} \pm z\frac{s}{\sqrt{n}}$$

$\bar{x}$ = mean classifier accuracy
z = confidence level value
s = standard deviation (computed above)
n = number of samples (10)
For the 95% confidence interval, z was set to 1.96.

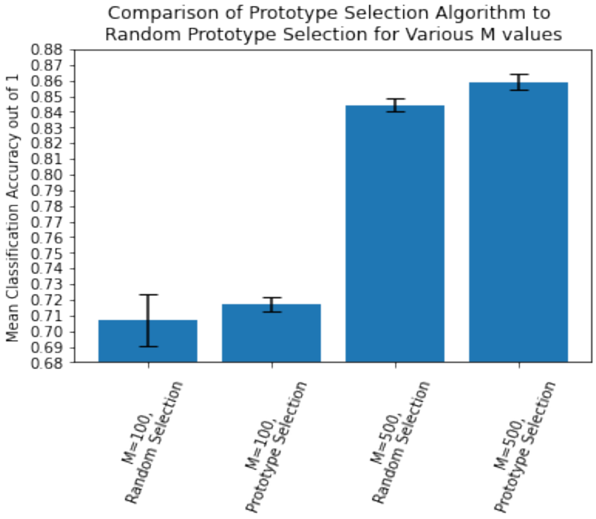


	Accuracy	95% Confidence	Lower Bound	Upper Bound
M=100, Random Selection	0.70673	0.01647	0.69026	0.72320
M=100, Prototype Algorithm	0.71738	0.00477	0.71261	0.72215
M=500, Random Selection	0.84452	0.00457	0.83995	0.84909
M=500, Prototype Algorithm	0.85898	0.00502	0.85396	0.86400
M=1000, Random Selection	0.88710	0.00173	0.88537	0.88883
M=1000, Prototype Algorithm	0.90160	0.00177	0.89983	0.90337
M=5000, Random Selection	0.93572	0.00103	0.93469	0.93675
M=5000, Prototype Algorithm	0.94325	0.00102	0.94223	0.94427
M=10000, Random Selection	0.94851	0.00144	0.94707	0.94995
M=10000, Prototype Algorithm	0.95017	0.00093	0.94924	0.95110

Table of Classifier Accuracy with error margins

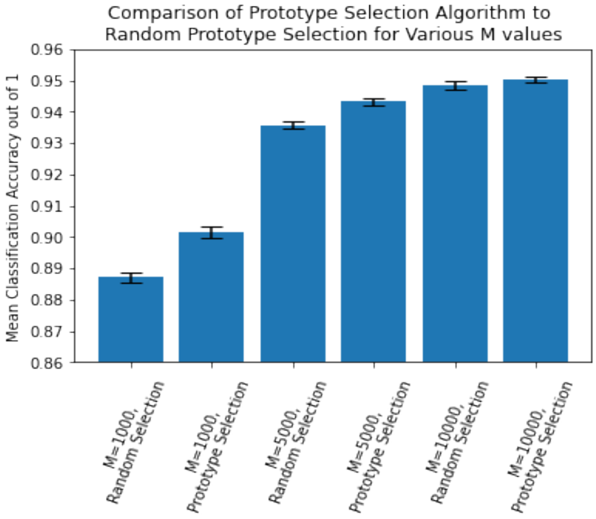
The table above includes the mean accuracy for the 10 experiments of each classifier type, the single-direction margin of error (labeled 95% confidence interval), and the upper and lower confidence interval bounds.

The table was produced via the Dataframe function of the Pandas library. It represents the numerical results completely, but what's a paper without matplotlib? Below you will find charts of the classifier accuracy data.



Random Prototype Selection vs Selection using our Algorithm for low M values

In this chart courtesy of the Matplotlib library, it is clear that the prototype selection algorithm gains 1.1% classification accuracy over random selection for M=100 and 1.4% classification accuracy for M=500. For M=100, the prototype selection algorithm significantly decreases the total margin of error of 3.30% down to just 0.94%.



Random Prototype Selection vs Selection using our Algorithm for high M values

This chart highlights that for the medial M values of 1,000 and 5,000, the selection algorithm outperforms random selection. For M = 1,000, the algorithm outperforms random selection by 1.45%, and for M=5000, the algorithm nets 0.75% accuracy. However as accuracy approaches the mid-90% range, the gains made by the algorithm become less significant. This is the case at M=10,000, where the 95% confidence intervals for random selection and the selection algorithm overlap and the overall accuracy differs by just 0.15%.

Although the Matplotlib charts illustrate how the prototype selection algorithm improves accuracy when compared to random prototype selection, the biggest take away from them is also the most obvious: The easiest way to increase the accuracy of a classifier is to increase the number of prototypes in the training set. As the size of M increases, the classification accuracy increases with it.

4 Critical Evaluation

In conclusion, judging strictly based on classifier accuracy, this algorithm for prototype selection is a clear improvement over random prototype selection for most values of M.

In particular, for the M values of 500, 1000, and 5000, the algorithm was clearly superior, gaining 1% classification accuracy on average. For all of these values, the lower bound of the margin of error for the selection algorithm exceeded the upper bound of the margin error for random selection. This suggests that the selection algorithm outperformed random selection at least 95% of the time for these M values.

For M=100, the algorithm decreased the margin of error considerably, making the classifier's accuracy much more consistent. However, it's hard to compare this classifier's total accuracy score to the highly inconsistent nature of a classifier trained with just 100 randomly selected prototypes.

On the other end at M=10,000, the gain in accuracy of the algorithm is marginal at best. This being said, it's challenging to improve on nearly 95% accuracy. If the number of prototypes is this large, one may as well choose them randomly.

Although this prototype selection algorithm is successful in increasing classifier accuracy, it has drawbacks. The most significant downside is the large run-time of this algorithm. Most operations are performed in linear time, however

because the algorithm is training and testing 50 classifiers before returning the final prototypes, the large number of computations leads to the long run-time.
The amount of time scales up for the algorithm to complete due to increased training time with greater M. At M=100, algorithm takes 1 minute to return the prototypes, yet can take up to 17 minutes for M=10,000.
This could be improved by decreasing the number of iterations in the algorithm, as the validation set accuracy appears to plateau before 50 iterations. This could significantly speed up the run-time without decreasing the accuracy of the classifier.

If this algorithm continued development, I would next like to incorporate nearest neighbor $distance$ as well as the identity of the nearest neighbor into the algorithm. For instance, within the set of nearest neighbor prototypes that led to correct predictions, it could be beneficial to evaluate the distance between the validation data point and its nearest neighbor and decide if that prototype is worth keeping based on the distance. Although this would also increase run-time, the gained accuracy from combining nearest neighbor distance and identity could make a difference for M values below 5,000.

Overall, this algorithm is effective for selecting prototypes to improve the 1-Nearest Neighbor classifier accuracy for a select range of M values.
At M below 100, it's very challenging to select the specific prototypes for this task, and the algorithm does not take the right approach to handle this few prototypes. At M exceeding 5000, this algorithm no longer provides any meaningful improvement over random selection and the run-time eclipses 10 minutes.
But for $100 < M < 5000$, this algorithm is very successful in that it almost guarantees a 1% increase in accuracy compared to random prototype selection with a slightly longer but still reasonable computational complexity.

Project 1: Prototype selection for 1-NN

```
In [725]: import gzip
          import numpy as np
          import tensorflow as tf
          import pandas as pd
          from sklearn.neighbors import KNeighborsClassifier
          import random
          import matplotlib.pyplot as plt
          %matplotlib inline
          import pandas as pd
          import plotly.figure_factory as ff
          import dataframe_image as dfi
```

Loading dataset and getting accuracy when using all training data

```
In [10]: (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```
In [13]: samples, nx, ny = x_train.shape
```

```
In [14]: x_trainReshape = x_train.reshape(samples, nx*ny)
```

```
In [15]: y_train.shape

Out[15]: (60000,)
```

```
In [16]: testsamples, testx, testy = x_test.shape
         x_testReshape = x_test.reshape(testsamples, testx*testy)
```

```
In [17]: y_test.shape

Out[17]: (10000,)
```

```
In [18]: baseline = KNeighborsClassifier(n_neighbors=1)
         baseline.fit(x_trainReshape, y_train)

Out[18]: KNeighborsClassifier(n_neighbors=1)
```

```
In [19]: baseline.score(x_testReshape, y_test)

Out[19]: 0.9691
```

Baseline 1-NN Score with all training data: 96.91%

Random prototype selection

In [20]:

```
xytrain = list(zip(x_trainReshape, y_train))
```

In [560]:

```
def randomPrototypeSelection(trainset, M):
    trainX, trainY = getRandomSamples(trainset, M)
    clf = KNeighborsClassifier(n_neighbors=1)
    clf.fit(trainX, trainY)
    score = clf.score(x_testReshape, y_test)
    return score
```

In [563]:

```
def randomExperiments(number, M):
    test_scores = []
    for i in range(number):
        exp = i+1
        #print(f"Experiment #{exp}")
        scoreTest = randomPrototypeSelection(xytrain, M)
        test_scores.append(scoreTest)
    return test_scores
```

Prototype selection algorithm

In [303]:

```
#returns indexes of all the correct predictions
def correctIndex(predictionY, actual):
    correctY = []
    for it, guess in enumerate(predictionY):
        #print(f"true value: {ytrain11[it]}, guess: {guess}")
        if guess == actual[it]:
            correctY.append(it)
    correctY = set(correctY)
    return correctY
```

In [524]:

```
def getXandY(correct,trainX,trainY, testX, testY, model):
    X = []
    Y = []
    for i in correct:
        #print(f"correct index: {i}")
        X.append(testX[i])
        Y.append(testY[i])
    neighbors = model.kneighbors(X, return_distance=False)
    goodProtoX = []
    goodProtoY = []
    neighbors = list(np.concatenate(neighbors).flat)
    neighbors = set(neighbors)
    for i in neighbors:
        xval = trainX[int(i)]
        yval = trainY[int(i)]
        goodProtoX.append(xval)
        goodProtoY.append(yval)

    return goodProtoX, goodProtoY
```

In [258]:
```
def getRandomSamples(trainSet, size):
    samples = random.sample(trainSet,size)
    xtrain = [i[0] for i in samples]
    ytrain = [i[1] for i in samples]
    return xtrain, ytrain
```

In [236]:
```
def concatenate(oldX, oldY, newX, newY):
    newX.extend(oldX)
    newYa = np.concatenate((newY, oldY), axis=None)
    return newX, newYa
```

In [426]:
```
def newClassifier(trainX, trainY, dataset, testsize):
    vsetX, vsetY = getRandomSamples(dataset,testsize)
    clf = KNeighborsClassifier(n_neighbors=1)
    clf.fit(trainX, trainY)
    preds = clf.predict(vsetX)
    score = clf.score(vsetX, vsetY)
    return preds, vsetX, vsetY, score, clf
```

In [521]:
```
def prototypeSelect(M, trainset, thresholdScore, totalIts):
    xtrain, ytrain = getRandomSamples(trainset, M)
    initPred,validX, validY, initScore, clf = newClassifier(xtrain, ytrain,

    score, protoX, protoY = recursion(initScore,0,xtrain,ytrain, initPred,v

    return score, protoX, protoY
```

In [570]:
```
ef recursion(score, iteration, xtrain, ytrain, ypred, validX, validY, train

   correct = correctIndex(validY, ypred)
   oldX, oldY = getXandY(correct, xtrain, ytrain, validX, validY, clf)
   #print(len(oldX))
   newX, newY = getRandomSamples(trainset,M-len(oldX))
   concatX, concatY = concatenate(newX, newY, oldX, oldY)
   preds, validX, validY, score, clf = newClassifier(concatX, concatY,xytra

   if iteration % 25 == 0 and iteration != totalIts:
       print(f"Iteration: {iteration} Score: {score}")

   if score >= thresholdScore or iteration >= totalIts:
       print(f"Final Iteration: {iteration} Final Score: {score}")
       return score, concatX, concatY
   else:
       iteration += 1
       score, concatX, concatY = recursion(score, iteration, concatX, conca

   return score, concatX, concatY
```

In [541]:
```
def testClassifier(protoX, protoY, testX, testY):
    clf = KNeighborsClassifier(n_neighbors=1)
    clf.fit(protoX, protoY)
    preds = clf.predict(testX)
    score = clf.score(testX, testY)
    print(f"Test Set Accuracy: {score}")
    return preds, score
```

In [549]:
```
def experiments(number, M):
    test_scores = []
    for i in range(number):
        exp = i+1
        print(f"Experiment #{exp}")
        scoreTrain, protoX, protoY = prototypeSelect(M, xytrain, 0.98, 50)
        predsTest, scoreTest = testClassifier(protoX, protoY, x_testReshape
        test_scores.append(scoreTest)
    return test_scores
```

In [550]:
```
testscores1000 = experiments(5,1000)
```

```
Experiment #1
Iteration: 0 Score: 0.8836
Iteration: 10 Score: 0.8893
Iteration: 20 Score: 0.893
Iteration: 30 Score: 0.8964
Iteration: 40 Score: 0.894
Final Iteration: 50 Final Score: 0.9035
Test Set Accuracy: 0.9038
Experiment #2
Iteration: 0 Score: 0.8817
Iteration: 10 Score: 0.889
Iteration: 20 Score: 0.8946
Iteration: 30 Score: 0.8952
Iteration: 40 Score: 0.8935
Final Iteration: 50 Final Score: 0.8929
Test Set Accuracy: 0.9016
Experiment #3
Iteration: 0 Score: 0.8835
Iteration: 10 Score: 0.8937
Iteration: 20 Score: 0.8948
```

In [575]:
```
testscores500_2 = experiments(5,500)
```

```
Experiment #1
Iteration: 0 Score: 0.8529
Iteration: 25 Score: 0.8541
Final Iteration: 50 Final Score: 0.8454
Test Set Accuracy: 0.8562
Experiment #2
Iteration: 0 Score: 0.8498
Iteration: 25 Score: 0.8578
Final Iteration: 50 Final Score: 0.8644
Test Set Accuracy: 0.8723
Experiment #3
Iteration: 0 Score: 0.8497
Iteration: 25 Score: 0.8516
Final Iteration: 50 Final Score: 0.8519
Test Set Accuracy: 0.8594
Experiment #4
Iteration: 0 Score: 0.8362
Iteration: 25 Score: 0.8492
Final Iteration: 50 Final Score: 0.847
Test Set Accuracy: 0.8544
```

In [576]:
```
random100_2 = randomExperiments(10, 100)
random500_2 = randomExperiments(10,500)
random1000_2 = randomExperiments(10,1000)
random5000_2 = randomExperiments(10,5000)
random10000_2 = randomExperiments(10,10000)
```

In [577]:
```
def standard_deviation(values):
    mean = np.mean(values)
    N = len(values)-1
    summ = 0
    for i in values:
        summ += np.square(i-mean)
    return np.sqrt(summ/N)
```

In [628]:
```
def confidence(values, conf_percent):
    z = {80: 1.282, 85:1.440, 90:1.645, 95: 1.960, 99:2.576, 99.5:2.807,99.
    if conf_percent not in z.keys():
        print("Please use a different confidence level")
        return
    mean = np.mean(values)
    n = len(values)
    s = standard_deviation(values)
    interval = z[conf_percent]*(s/np.sqrt(n))
    lower = mean-interval
    upper = mean+interval
    #print(f"{conf_percent}% confidence interval: {mean} +/- {interval}")
    return mean, lower, upper, interval
```

In [594]:
```
testscores100.extend(testscores100_2)
```

In [596]:

```
testscores500.extend(testscores500_2)
testscores1000.extend(testscores1000_2)
testscores5000.extend(testscores5000_2)
testscores10000.extend(testscores10000_2)
```

In [668]:

```
mean100,mean100_lower,mean100_upper,int100 = confidence(testscores100,95)
```

In [669]:

```
meanrand100,meanrand100_lower,meanrand100_upper,intrand100 = confidence(ran
```

In [675]:

```
mean500, mean500_lower, mean500_upper,int500 = confidence(testscores500,95)
meanrand500, meanrand500_lower,meanrand500_upper,intrand500 = confidence(ra
mean1000,mean1000_lower,mean1000_upper,int1000 = confidence(testscores1000,
meanrand1000,meanrand1000_lower,meanrand1000_upper,intrand1000 = confidence
mean5000,mean5000_lower,mean5000_upper,int5000 = confidence(testscores5000,
meanrand5000,meanrand5000_lower,meanrand5000_upper,intrand5000 = confidence
mean10000,mean10000_lower,mean10000_upper,int10000 = confidence(testscores1
meanrand10000, meanrand10000_lower,meanrand10000_upper,intrand10000 = confi
```

In [727]:

```
means = [meanrand100, mean100,meanrand500, mean500,meanrand1000, mean1000,m
lower = [meanrand100_lower, mean100_lower,meanrand500_lower,mean500_lower,m
upper = [meanrand100_upper,mean100_upper,meanrand500_upper,mean500_upper,me
interval = [intrand100, int100,intrand500, int500,intrand1000, int1000,intr
labels = ["M=100, Random Selection", "M=100, Prototype Algorithm", "M=500,
```

In [728]:

```
data = {'Accuracy':means,'95% Confidence':interval,'Lower Bound':lower, 'Up
```

In [729]:

```
df = pd.DataFrame(data, index=labels )
df2 = df.round(5)
df2
```

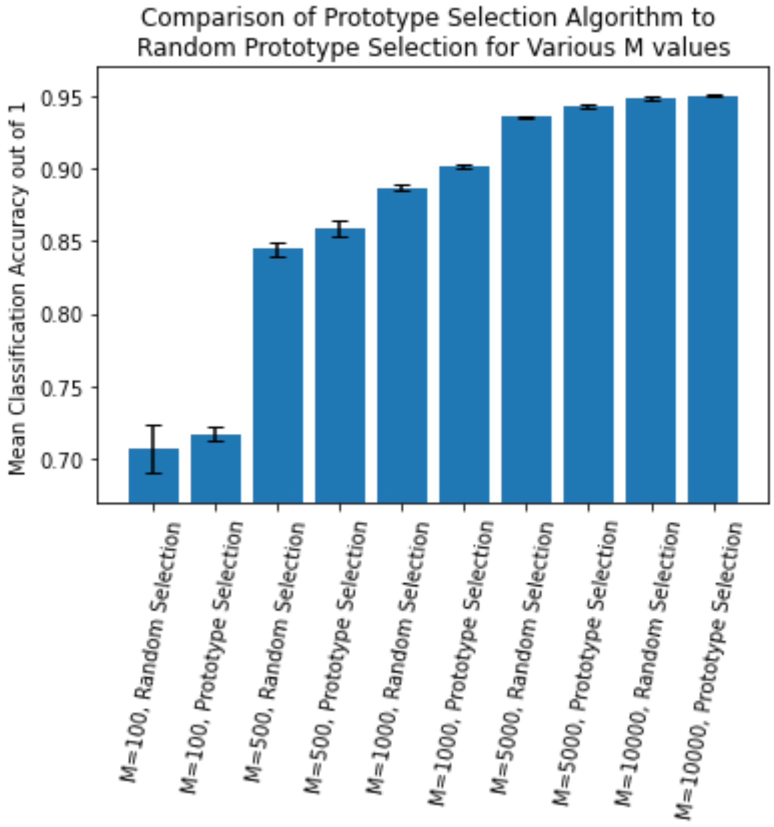
Out[729]:

	Accuracy	95% Confidence	Lower Bound	Upper Bound
M=100, Random Selection	0.70673	0.01647	0.69026	0.72320
M=100, Prototype Algorithm	0.71738	0.00477	0.71261	0.72215
M=500, Random Selection	0.84452	0.00457	0.83995	0.84909
M=500, Prototype Algorithm	0.85898	0.00502	0.85396	0.86400
M=1000, Random Selection	0.88710	0.00173	0.88537	0.88883
M=1000, Prototype Algorithm	0.90160	0.00177	0.89983	0.90337
M=5000, Random Selection	0.93572	0.00103	0.93469	0.93675
M=5000, Prototype Algorithm	0.94325	0.00102	0.94223	0.94427
M=10000, Random Selection	0.94851	0.00144	0.94707	0.94995
M=10000, Prototype Algorithm	0.95017	0.00093	0.94924	0.95110

In [730]:
```
dfi.export(df2, 'PandasTable2.png')
```

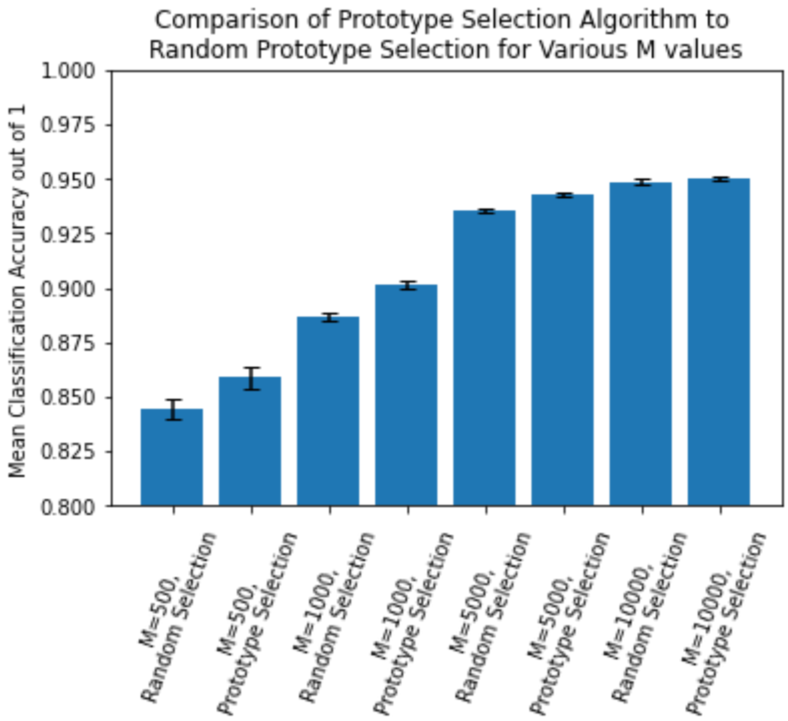
In [737]:
```
y = [meanrand100, mean100,meanrand500, mean500,meanrand1000, mean1000,meanr
x = ["M=100, Random Selection", "M=100, Prototype Selection", "M=500, Rando
y_error = [intrand100, int100,intrand500, int500,intrand1000, int1000,intra
plt.bar(x,y)
plt.title("Comparison of Prototype Selection Algorithm to \nRandom Prototyp
plt.ylabel("Mean Classification Accuracy out of 1")
plt.errorbar(x,y,yerr=y_error,fmt='none', barsabove=True, ecolor='black',ca
plt.xticks(rotation=80)
plt.ylim([0.67,0.97])
```

Out[737]:
```
(0.67, 0.97)
```



```
In [736]: y = [meanrand500, mean500,meanrand1000, mean1000,meanrand5000, mean5000,mea
          x = [ "M=500,\nRandom Selection", "M=500,\nPrototype Selection","M=1000,\nR
          y_error = [intrand500, int500,intrand1000, int1000,intrand5000, int5000,int
          plt.bar(x,y)
          plt.title("Comparison of Prototype Selection Algorithm to \nRandom Prototyp
          plt.ylabel("Mean Classification Accuracy out of 1")
          plt.xticks(rotation=70)
          plt.errorbar(x,y,yerr=y_error,fmt='none', barsabove=True, ecolor='black')
          plt.ylim([0.8,1])
```

```
Out[736]: (0.8, 1.0)
```

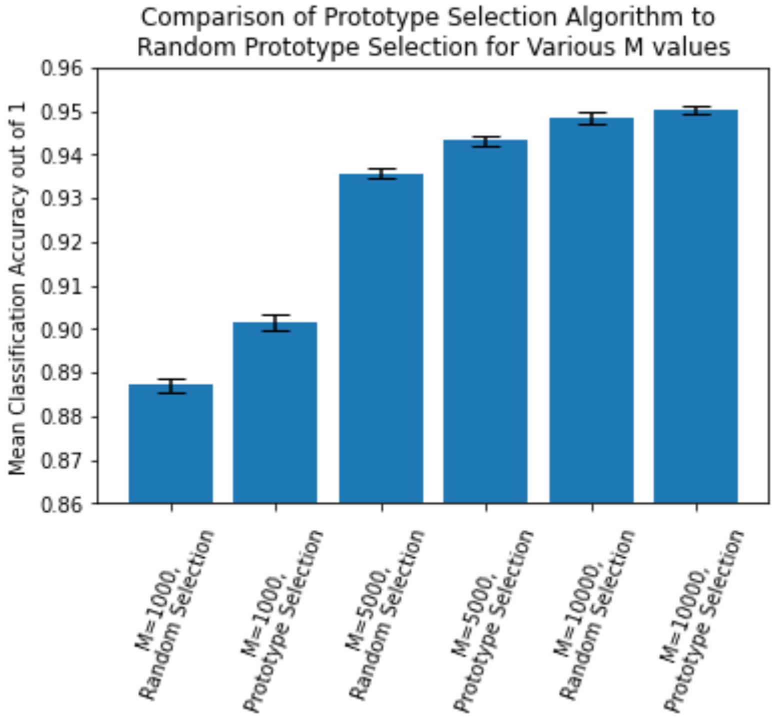


In [751]:

```
y = [meanrand1000, mean1000,meanrand5000, mean5000,meanrand10000, mean10000
x = [ "M=1000,\nRandom Selection", "M=1000,\nPrototype Selection","M=5000,\
y_error = [intrand1000, int1000,intrand5000, int5000,intrand10000, int10000
plt.bar(x,y)
plt.title("Comparison of Prototype Selection Algorithm to \nRandom Prototyp
plt.ylabel("Mean Classification Accuracy out of 1")
plt.xticks(rotation=70)
plt.yticks(np.arange(0.85,max(y)+1,0.01))
plt.errorbar(x,y,yerr=y_error,fmt='none', barsabove=True, ecolor='black',ca
plt.ylim([0.86,0.96])
```

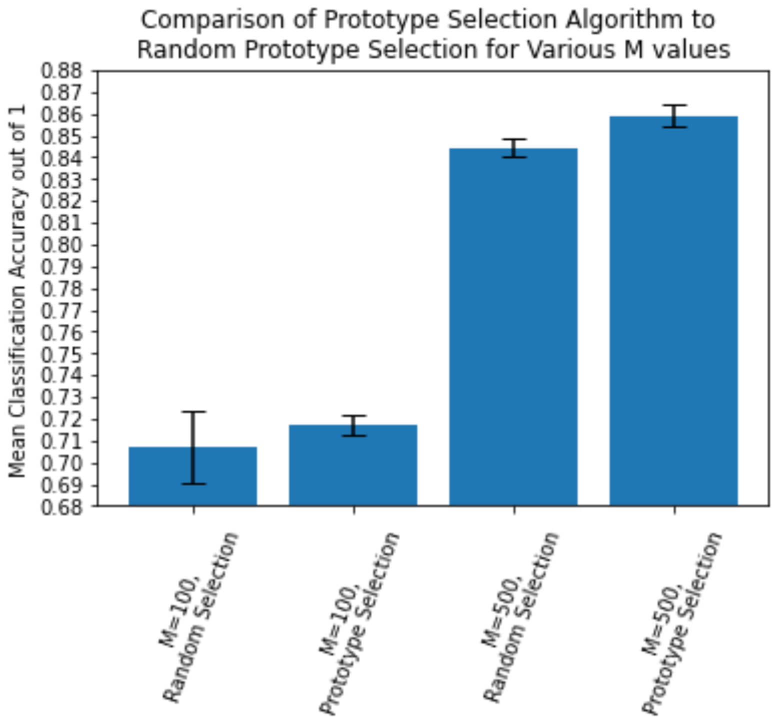
Out[751]:

```
(0.86, 0.96)
```



In [745]:
```
y = [meanrand100, mean100,meanrand500, mean500]
x = [ "M=100,\nRandom Selection", "M=100,\nPrototype Selection","M=500,\nRa
y_error = [intrand100, int100,intrand500, int500]
plt.bar(x,y)
plt.title("Comparison of Prototype Selection Algorithm to \nRandom Prototyp
plt.ylabel("Mean Classification Accuracy out of 1")
plt.xticks(rotation=70)
plt.yticks(np.arange(0.66,max(y)+1,0.01))
plt.errorbar(x,y,yerr=y_error,fmt='none', barsabove=True, ecolor='black',ca
plt.ylim([0.68,0.88])
```

Out[745]:
```
(0.68, 0.88)
```


Comparison of Prototype Selection Algorithm to
Random Prototype Selection for Various M values
Mean Classification Accuracy out of 1
M=100,
Random Selection
M=100,
Prototype Selection
M=500,
Random Selection
M=500,
Prototype Selection