Mira Plante
Assignment 4
4/18/2024

PRNG Classification Report

Data Preparation

I originally spent much of my time trying to combine the datasets into one dataset for training, validation, and testing, but realized later on that it would be difficult to do so without severely impacting model accuracy. So, I went with creating 3 separate models, and with that, 3 separate sets of testing/training/validation data. **All of the data preparation steps are located in the full codebase (link).**

Overall Discussion

I was very confused when initially doing this assignment, as I could not get any CNN layering methods over 25.5%, but after looking at it from a security standpoint, this seems to be a point in favor of PRNG models, as it is difficult to successfully identify a PRNG based on its numerical output.

Convolutional Neural Network (CNN)

For the CNN, I wanted to experiment with different layering methods I had not tried before, so I used the Conv1D and MaxPooling1D to see if that would help with compile time without impacting accuracy majorly (which was the case). I also tried adding different layers to see how they would impact both runtime and accuracy, however it seemed the base model I constructed had similar accuracy scores to all of the modified models I created. For the loss in the compilation portion, I knew it had to be categorical because of the way the datasets were constructed to be in "categories" 1-8, and I ended up choosing SparseCategoricalCrossEntropy because it gave the best results. I chose the Adam optimizer because it was what I was most familiar with out of all of the optimizers. When testing, I kept the epochs relatively low at 10 due to time constraints, but increasing the epochs could have helped boost the accuracy. Overall, I was surprised with how low the testing and training set accuracies were (**a maximum of 25.4% accuracy for testing set**), as they were comparable with some of the models below, but with a longer execution time. I believe there is an optimal layering that could be achieved to increase accuracy, however it still may not reach over 50% accuracy, so it would be no better than guessing.

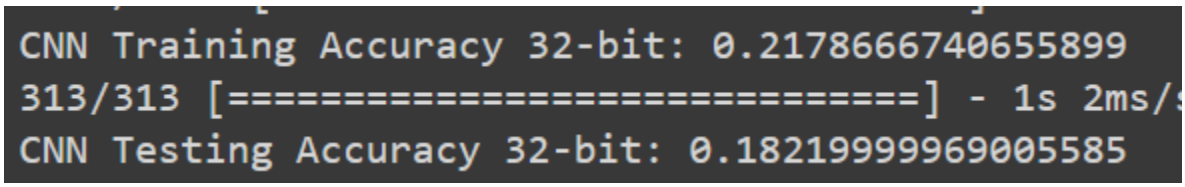
Below is the code, a screenshot output of the accuracies for each dataset size, and a table showing the information as well:

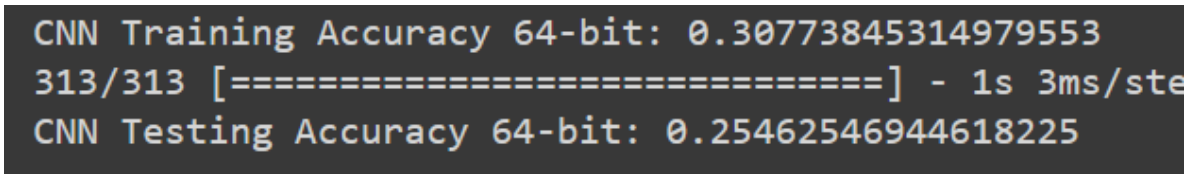
Code for optimizing hyperparameters and testing (copied from Colab because VS Code did not like my version of Tensorflow for some reason):

```
# CNN
# general structure from https://www.tensorflow.org/tutorials/images/cnn

# from
https://stackoverflow.com/questions/42928911/reshape-a-pandas-dataframe
rXTrain1 = xTrain1C.values.reshape(xTrain1C.shape[0], xTrain1C.shape[1],
1)
rXTest1 = xTest1.values.reshape(xTest1.shape[0], xTest1.shape[1], 1)
rXTrain2 = xTrain2C.values.reshape(xTrain2C.shape[0], xTrain2C.shape[1],
1)
rXTest2  = xTest2.values.reshape(xTest2.shape[0], xTest2.shape[1], 1)
rXTrain3 = xTrain3C.values.reshape(xTrain3C.shape[0], xTrain3C.shape[1],
1)
rXTest3 = xTest3.values.reshape(xTest3.shape[0], xTest3.shape[1], 1)

# 32-bit
# from https://www.tensorflow.org/api_docs/python/tf/keras/Sequential
```

```
# relu info from
https://www.tensorflow.org/api_docs/python/tf/keras/activations/relu
# other info from associated sites on the keras docs
model = Sequential([Conv1D(32, kernel_size=2, activation='relu',
input_shape=(rXTrain1.shape[1], 1)),
    MaxPooling1D(pool_size=5),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(9, activation='softmax')])

# loss info from
https://www.tensorflow.org/api_docs/python/tf/keras/losses
# optimizer info from https://keras.io/api/optimizers/
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.fit(rXTrain1, yTrain1C, validation_data=(xVal1, yVal1), epochs=10)

useless, accuracy = model.evaluate(rXTrain1, yTrain1C)
print("CNN Training Accuracy 32-bit:", accuracy)
useless, accuracy = model.evaluate(rXTest1, yTest1)
print("CNN Testing Accuracy 32-bit:", accuracy)
```

Screenshot of training and testing set accuracies for each dataset size:





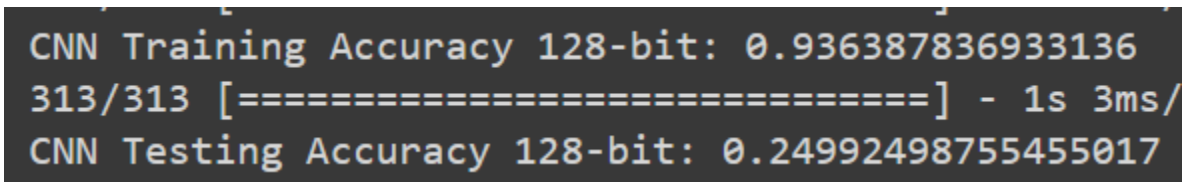


Table comparing the training and testing set accuracies for each dataset size, noticing how the 32-bit dataset always has the lowest accuracy:

Type	Dataset Size (bits)	Accuracy
Training	32	21.8%
	64	30.8%
	128	93.6%
Testing	32	18.2%
	64	25.5%
	128	25.0%

Support Vector Machine (SVM)

For the support vector machine (SVM) model, I used the scikit-learn GridSearchCV function to try to get the best accuracy, however it drastically lengthened execution time with virtually no improvement to the accuracy score, so I commented it out early on. With the accuracy score being similar to other models (**testing accuracy being 25.5% at its highest**) with an additional extremely long execution time, and hyperparameter tuning not modifying accuracy scores much at all, I would not consider this model viable for breaking PRNGs.

Below is the code, a screenshot output of the accuracies for each dataset size, and a table showing the information as well:

Code for optimizing hyperparameters and testing:

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

svm = svm.SVC()

# hyperparemeter tuning from
https://scikit-learn.org/stable/modules/grid_search.html
# but it didn't change accuracy and increased execution time so not
doing it! It seemed that 'rbf' was the best kernel,
# but it increased execution time even more than it already is so I left
it as the default version,
# as the accuracy score increase was not significant
#param_grid = {"C": [0.001, 0.01, 0.1, 1, 10, 100], "kernel": ["linear",
"poly", "rbf", "sigmoid"]}
#grid_search = GridSearchCV(estimator=lin, param_grid=param_grid,
scoring="accuracy")
#svm = grid_search

# Actually running the model (it's the same for 64/128 bit with
different sets for testing/training)
svm.fit(xTrain1, yTrain1)
model = svm.predict(xTrain1)
accuracy = accuracy_score(yTrain1, model)
print("SVM Training Accuracy 32-bit:", accuracy)

model = svm.predict(xTest1)
```

```
accuracy = accuracy_score(yTest1, model)
print("SVM Testing Accuracy 32-bit:", accuracy)
```

Screenshot of training and testing set accuracies for each dataset size:

```
SVM Training Accuracy 32-bit: 0.3893285714285714
SVM Testing Accuracy 32-bit: 0.1868
SVM Training Accuracy 64-bit: 0.576265375219646
SVM Testing Accuracy 64-bit: 0.255025502550255
SVM Training Accuracy 128-bit: 0.7352516752153848
SVM Testing Accuracy 128-bit: 0.2548254825482548
```

Table comparing the training and testing set accuracies for each dataset size, noticing how the 32-bit dataset always has the lowest accuracy, while 64-bit and 128-bit are similar:

Type	Dataset Size (bits)	Accuracy
Training	32	38.9%
	64	57.6%
	128	73.5%
Testing	32	18.7%
	64	25.5%
	128	25.5%

K-Nearest Neighbors (KNN)

For the k-nearest neighbors (KNN) model, I used a for loop to go through several ranges of possible nearest-neighbors k values to obtain optimal accuracy for 32, 64, and 128-bit inputs. For 32-bit, for example, it was determined that k=16 seemed to be an optimal value that produced the highest possible accuracy without taking too long to run through. I repeated this step with the other bit lengths as well. An example of this is in the code shown below for the model setup and execution. I also included a screenshot of the outputs of training and testing accuracy (there was no separate validation set, as the validation set became part of the training set) below along with a table of accuracies for each bit length and input set.

Even with optimization of the k value, even the training set accuracy the model was trained on **could not reach accuracy above 35%**, showing just how difficult it is to predict and break the listed PRNGs.

Code for optimizing number of neighbors and testing:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Optimizing to find the best number of neighbors
accuracy = 0
for i in range(1,26):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(xTrain1, yTrain1)
    model = knn.predict(xTest1)
    newAccuracy = accuracy_score(yTest1, model)
    if newAccuracy > accuracy:
        accuracy = newAccuracy
        print(str(i), " is the best!")

# Actually running the model (basically the same for 64/128-bit)
knn = KNeighborsClassifier(n_neighbors=16)

# 32-bit
knn.fit(xTrain1, yTrain1)
model = knn.predict(xTrain1)
accuracy = accuracy_score(yTrain1, model)
print("KNN Training Accuracy 32-bit:", accuracy)
model = knn.predict(xTest1)
accuracy = accuracy_score(yTest1, model)
```

```
print("KNN Testing Accuracy 32-bit:", accuracy)
```

Screenshot of training and testing set accuracies for each dataset size:

```
KNN Training Accuracy 32-bit: 0.306928571428571 44
KNN Testing Accuracy 32-bit: 0.1895
KNN Training Accuracy 64-bit: 0.3511050157859398
KNN Testing Accuracy 64-bit: 0.238423842384238 42
KNN Training Accuracy 128-bit: 0.350759383349287 75
KNN Testing Accuracy 128-bit: 0.23742374237423 74
```

Table comparing the training and testing set accuracies for each dataset size, noticing how the 32-bit dataset always has the lowest accuracy, while 64-bit and 128-bit are similar:

Type	Dataset Size (bits)	Accuracy
Training	32	30.7%
	64	35.1%
	128	35.0%
Testing	32	19.0%
	64	23.8%
	128	23.7%

Decision Tree (DT)

For the decision tree (DT) model, as it was able to run very quickly compared to some of the other models, I was more free to optimize parameters using the GridSearchCV function. Interestingly, the DT model had 100% accuracy in the training set without parameter optimization, but much significantly lower training set accuracy with the parameter optimization in place, though this makes sense with how DT models parse through information for training sets. The testing set accuracy for the 128-bit model was also lowered when parameters were optimized, with the training accuracy remaining at 100%. Even with these accuracy issues, the DT was able to **reach an accuracy score of 25.3%** for the testing set and considering the speed tradeoff when looking at the other models with similar or lower accuracies, DT may be a more viable option for predicting which PRNG generated a string of numbers.

I have included screenshots for both, along with the code for the DT model and the accuracy outputs below:

Code for optimizing parameters and testing, with 32-bit representing the code for the other datasets as well:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

dtc = DecisionTreeClassifier()

# parameter tuning, from above source and looking at DT parameters
param_grid = {'criterion': ['gini', 'entropy'], 'max_depth': [None, 5,
10, 15]}
grid_search = GridSearchCV(estimator=dtc, param_grid=param_grid,
scoring='accuracy')
dtc = grid_search

# 32-bit
dtc.fit(xTrain1, yTrain1)
model = dtc.predict(xTrain1)
accuracy = accuracy_score(yTrain1, model)
print("DT Training Accuracy 32-bit:", accuracy)

model = dtc.predict(xTest1)
accuracy = accuracy_score(yTest1, model)
print("DT Testing Accuracy 32-bit:", accuracy)
```

Screenshot of training and testing set accuracies for each dataset size:

```
DT Training Accuracy 32-bit: 1.0
DT Testing Accuracy 32-bit: 0.1845
DT Training Accuracy 64-bit: 1.0
DT Testing Accuracy 64-bit: 0.24892489248924893
DT Training Accuracy 128-bit: 1.0
DT Testing Accuracy 128-bit: 0.24902490249024903
```

```
DT Training Accuracy 32-bit: 0.2872857142857143
DT Testing Accuracy 32-bit: 0.1859
DT Training Accuracy 64-bit: 0.896298518550265
DT Testing Accuracy 64-bit: 0.253325332533253334
DT Training Accuracy 128-bit: 1.0
DT Testing Accuracy 128-bit: 0.24772477247724772
```

Table comparing the training and testing set accuracies for each dataset size for the optimized model, noticing how the 32-bit dataset always has the lowest accuracy:

Type	Dataset Size (bits)	Accuracy
Training	32	28.7%
	64	89.7%
	128	100.0%
Testing	32	18.6%
	64	25.3%
	128	24.8%

Logistic Regression

I chose the logistic regression model because I wanted to see how a very rigid model would interpret the PRNG outputs. For the model, I tuned it using the hyperparameter tuning method used in previous methods, specifically on the C parameter. I also played with modifying the penalty, but this resulted in some errors thrown by the model about infinite values (possibly due to interactions with the C parameter), so I left it at modification of the C parameter. This did not improve accuracy scores, only increasing execution time, however the model was still much faster than the other models given. However, as expected, the performance is abysmal, with **2% being the highest testing accuracy**. Interestingly enough, the 32-bit dataset had the highest testing and training accuracy, which is opposite of what happened with the rest of the models.

I have included screenshots for both, along with the code for the DT model and the accuracy outputs below:

Code for optimizing parameters and testing, with 32-bit representing the code for the other datasets as well:

```
from sklearn.metrics import log_loss
from sklearn.model_selection import GridSearchCV
from sklearn import linear_model

lin = linear_model.LogisticRegression(max_iter=1000)

# hyperparemeter tuning from
https://scikit-learn.org/stable/modules/grid_search.html
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}
grid_search = GridSearchCV(estimator=lin, param_grid=param_grid,
scoring='accuracy')
lin = grid_search

# 32-bit
lin.fit(xTrain1, yTrain1)
model = lin.predict_proba(xTrain1)
accuracy = log_loss(yTrain1, model)
print("Logistic Regression Training Accuracy 32-bit:", accuracy)

model = lin.predict_proba(xTest1)
accuracy = log_loss(yTest1, model)
print("Logistic Regression Testing Accuracy 32-bit:", accuracy)
```

Screenshot of training and testing set accuracies for each dataset size:

```
Logistic Regression Training Accuracy 32-bit: 1.9758794131120576
Logistic Regression Testing Accuracy 32-bit: 1.9814830943316581
Logistic Regression Training Accuracy 64-bit: 1.6999353142150995
Logistic Regression Testing Accuracy 64-bit: 1.7073953392265657
Logistic Regression Training Accuracy 128-bit: 1.6974143871239338
Logistic Regression Testing Accuracy 128-bit: 1.7063172494442762
```

Table comparing the training and testing set accuracies for each dataset size for the optimized model, noticing how the 32-bit dataset has the highest accuracy and how training and testing sets have the same rounded accuracies:

Type	Dataset Size (bits)	Accuracy
Training	32	2%
	64	1.7%
	128	1.7%
Testing	32	2%
	64	1.7%
	128	1.7%