PROJECT REPORT:
ANALYSIS ON GUN OWNERSHIP CORRELATION ON GUN DEATHS

Team Members: Teresa Chu

I. PROBLEM

The right to own a gun is an important right of freedom for Americans. Allowing people to easily access guns allows for crimes such as mass shootings in public areas to occur in increasing frequency over the last decades. However, gun ownership plays a significant role for people to protect property and family or to participate in activities like hunting. For this reason, increasing regulations on gun ownership in America and preventing people from easily owning a gun has been widely debated.

Theoretically, investigation on the gun owner's background can be used to see if there is a likelihood that a crime may be committed by the future gun owner. If it is possible to predict the deaths that will occur from background checks, increasing regulations on gun ownership may have a positive impact and prevent shootings from occurring. Therefore, this project aims to see if there is a correlation between background checks conducted and deaths in America. If background check attributes can be used to predict the number of deaths using logistic regression and SVM models, it may signify those regulations can be further looked into as a method to prevent gun deaths from occurring.

In order to find this correlation, two datasets will be used for this project will be from public datasets. One dataset used is background checks on gun ownership processed from the FBI National Instant Criminal Background Check System[1]. Another dataset will be used on gun deaths in America from the data.world public dataset site[2].

II. SOFTWARE DESIGN AND IMPLEMENTATION

A. Software Design and NoSQL Database and Tools Used

Key tools
For this project, the NoSQL database implemented was Apache Spark in python. In order to use this database, this was implemented using the findspark and pyspark modules. The environment that Apache Spark was used in is Jupiter Notebook. Several libraries that were used in this project include pandas, numpy, pyspark.ml.classification, pyspark.ml.evaluation and pyspark.ml.feature.

Software Design
Given two datasets, these datasets must first be cleaned and combined. For the background check dataset, statistics are divided by state per month and information on several background check attributes before 1999 is either unavailable or inconsistent. The other dataset regarding deaths is divided by year per county and information past 2018 is not available. As such, both datasets were cleaned to have information per state and per year and to have information from 1999-2018.

After these datasets were cleaned and combined, logistic regression and SVM models were used in order to make predictions on the number of deaths per state per year based on the background check attribute information. Given that these models make these predictions based off on a binary classification, the number of deaths for the state were classified as either greater than or less than the average number of deaths in the two-decade period. The data for each state was then randomly divided into two sets for the training of the model and for actual prediction test.

B. Implementation

Models used for the analysis of this project include logistic regression models and SVM models. Both models were created for each state and used to classify if the gun deaths in the year were above or below average from the attributes given. For the SVM model, a random initial weight was set for each feature from -0.1 to 0.1. Then after several iterations, the weight for that particular feature was updated towards the actual outcome. Prediction results were then printed out and drawn as a bar graph through matplotlib library. The results from this model were then shown for each region in the US to see if there was a correlation between these results and the region where the state came from. In addition, this was then compared to a logistic regression model created through pyspark's machine library and graphed out.

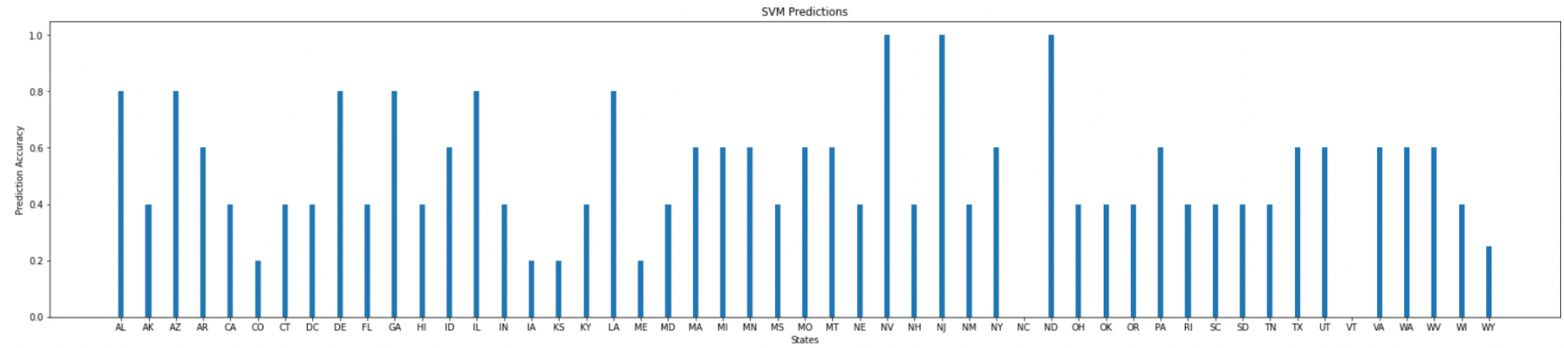
The self-implemented logistic regression model used the standard model. An initial weight of 0 was used for each feature which was then updated based off the sigmoid function that the logistic regression model is based off of. In order to prevent programming errors from occurring due to the sigmoid function being out of range, values in the dataset were normalized. After several iterations, the resulting weight and bias were used to make predictions off of the test dataset. The accuracy for each state was calculated and then printed out. This was compared with pyspark's logistic regression model through matplotlib.

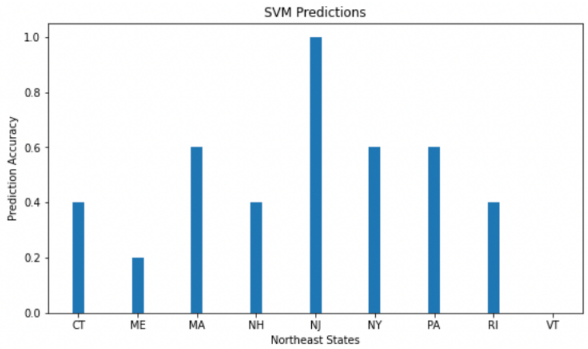
III. PROJECT OUTCOME

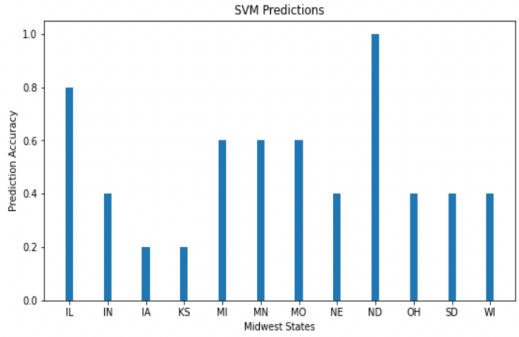
Results for this project indicated an overall low accuracy in predicting gun deaths from the background check attributes. This may be an indication that these background check attributes have little or no correlation with gun deaths. However, these results can be attributed to the dataset used. The background check dataset comes from official sales of guns which does not account for private sales and trades of guns. Moreover, there was a limited selection on background check attributes to be used in this dataset. As such, there could still be a possibility there is a correlation between background checks and gun deaths.

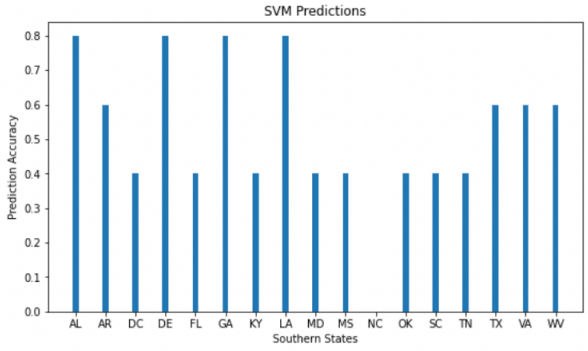
When looking at the SVM results, the overall low accuracy is the result from being skewed from the low accuracy level predictions of several states. The region from where the state comes from appears to have little correlation with the accuracy levels as well.

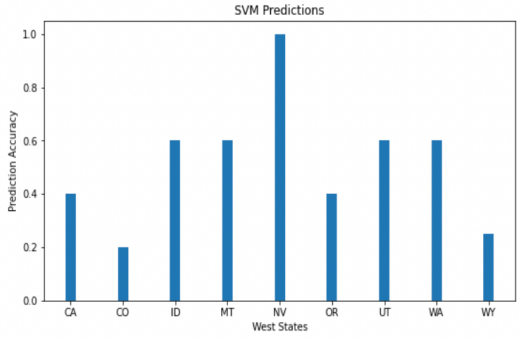
The only region where there appears to have some correlation between background checks and gun deaths is the states in the southern region of the US. However, even then, there are still several states within that region with a low accuracy in prediction results.











The logistic regression model has similar results to the SVM models. As a result, there seems to be little correlation with background checks and gun deaths. As such, instead of looking further into background checks, different avenues to limit gun death can be looked into. Examples of this could include further regulations on bullet sales or regulations on specific types of guns.

V. Source Code

```
import findspark
findspark.init('Users/teresachu/server/spark-2.4.3-bin-hadoop2.7')
import pyspark
```

```
import matplotlib.pyplot as plt


BCpath = "BackgroundChecks.csv"
GDpath = "GunDeaths.csv"
states = ["AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DC", "DE",
"FL", "GA","HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME",
"MD",
          "MA", "MI", "MN", "MS", "MO", "MT", "NE", "NV", "NH",
"NJ","NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC",
          "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI",
"WY"]


df =
spark.read.options(inferSchema='True',header='True').csv(BCpath)
df2 =
spark.read.options(inferSchema='True',header='True').csv(GDpath)

df.printSchema()


#For each state get all information from 1999-2008 for background
checks done per year instead of month
stateinfo = list()
df = df.drop("month","state full name")
bdf =
df.groupBy("year","state").sum("permit","permit_recheck","long_gun"
,"handgun","multiple","other","totals").orderBy("state")
for state in states:
    stateyear = bdf.filter((bdf.state == state))
    stateinfo.append(stateyear)

for stateinfos in stateinfo:
    stateinfos.show()


#For each state get all information from 1999-2008 for gun deaths
done
df2.printSchema()


#Get sum of gun deaths per state instead of county
gundeathinfo = list()
gdf = df2.groupBy("year","state").sum("Deaths","Population") \
        .withColumnRenamed("sum(Deaths)", "Deaths") \
        .withColumnRenamed("sum(Population)",
"Population").orderBy("state")
for state in states:
    stateyear = gdf.filter((gdf.state == state))
```

```
    statepercent = stateyear.withColumn("deathpercent",
100*(stateyear.Deaths/stateyear.Population))
    statepercent.show()
    gundeathinfo.append(statepercent)


newgundeathinfo = []
for gunstate in gundeathinfo:
    avgdeath =
(gunstate.groupBy("state").sum("deathpercent").collect()[0][1])/gun
state.count()

    #Add outcome col called above average as 0 or 1
    newgunstate = gunstate.withColumn("label",
(gunstate.deathpercent > avgdeath).cast("int"))
    newgundeathinfo.append(newgunstate)

# Join the attributes of the background chck with the death
information to make new table
stateCDF = []
for i in range(len(stateinfo) ):
    combined =
stateinfo[i].join(newgundeathinfo[i],['state','year'])
    stateCDF.append(combined)

for table in stateCDF:
    table.show()

from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.feature import VectorAssembler
import pandas as pd
import math
import statistics

def getStatistics(column):
    mean = sum(column)/len(column)
    std = statistics.pstdev(column)
    return mean,std

#Logistic Regression anaylsis using pyspark library
#To show usage of apache spark
predictionState = []
for table in stateCDF:
    #STATE NAME
    statename = table.select("state").first()[0]

    #Get info to pandas to do mlib
    #Remove attributes that will not be feature or outcome
```

```
    newtable =
table.drop("state","year","Deaths","Population","deathpercent")

    #Get all features that can be attributes which is everything
except above average
    features = []
    for feature in newtable.dtypes:
        if feature[1] != 'int':
            features.append(feature[0])

    #Use vector assembler for logistic regression prediction
    vectorass = VectorAssembler(inputCols = features,
outputCol='features')
    vectorDF = vectorass.transform(newtable)
    vectorDF = vectorDF.select(['features', 'label'])
    #Split into training and test data
    train, test = vectorDF.randomSplit([0.7, 0.3])

    print("train data")
    train.show()
    print("test data")
    test.show()

    #Run Logistic Regression on data
    logReg = LogisticRegression(featuresCol = 'features', labelCol
= 'label', maxIter=6)
    trainedModel = logReg.fit(train)
    prediction = trainedModel.transform(test)

    print("prediction")
    prediction.show()
    #Evaluate data

    evaluator = BinaryClassificationEvaluator()
    accuracy =  evaluator.evaluate(prediction)
    predictionState.append([statename, accuracy])


overallaccuracy = 0
for prediction in predictionState:
    print("State: ",prediction[0], " Accuracy: ", prediction[1])
    if (prediction[1] > 0.85):
        print("Current Background Checks indicator on deaths \n")
    else:
        print("Avenues other than background checks can be
considered or increase background checks\n")
    overallaccuracy += prediction[1]
print("Overall Accuracy: ",overallaccuracy/50)
```

```
#Graph results
xlabel = [prediction[0] for prediction in predictionState]
yvalues = [prediction[1] for prediction in predictionState]

fig = plt.figure(figsize = (30, 6))
plt.bar(xlabel,yvalues,width = 0.2)

plt.xlabel("States")
plt.ylabel("Prediction Accuracy")
plt.title("Logistic Regression Predictions")
plt.show()


#Get accuracies for states in each region
north = ["PA","NY","VT","ME","NH","MA","CT","RI","NJ"]
midwest =
["ND","SD","NE","MN","KS","MO","IA","WI","IL","IN","MI","OH"]
south =
["TX","OK","AR","LA","MS","AL","GA","FL","TN","SC","NC","KY","VA","
WV","DC","DE","MD"]
west = ["WA","CA","UT","CO","NV","OR","MT","ID","WY"]

# Creating the bar plot for the northern states
Nxlabel = [prediction[0] for prediction in predictionState if
prediction[0] in north]
Nyvalues = [prediction[1] for prediction in predictionState if
prediction[0] in north]
fig = plt.figure(figsize = (9, 5))
plt.bar(Nxlabel,Nyvalues,width = 0.2)
plt.xlabel("Northeast States")
plt.ylabel("Prediction Accuracy")
plt.title("Logistic Regression Predictions")
plt.show()

# Creating the bar plot for the south states
Sxlabel = [prediction[0] for prediction in predictionState if
prediction[0] in south]
Syvalues = [prediction[1] for prediction in predictionState if
prediction[0] in south]
fig = plt.figure(figsize = (9, 5))
plt.bar(Sxlabel,Syvalues,width = 0.2)
plt.xlabel("Southern States")
plt.ylabel("Prediction Accuracy")
plt.title("Logistic Regression Predictions")
plt.show()

# Creating the bar plot for the Midwest
```

```
Mxlabel = [prediction[0] for prediction in predictionState if
prediction[0] in midwest]
Myvalues = [prediction[1] for prediction in predictionState if
prediction[0] in midwest]
fig = plt.figure(figsize = (9, 5))
plt.bar(Mxlabel,Myvalues,width = 0.2)
plt.xlabel("Midwest States")
plt.ylabel("Prediction Accuracy")
plt.title("Logistic Regression Predictions")
plt.show()

# Creating the bar plot for the West
Wxlabel = [prediction[0] for prediction in predictionState if
prediction[0] in west]
Wyvalues = [prediction[1] for prediction in predictionState if
prediction[0] in west]
fig = plt.figure(figsize = (9, 5))
plt.bar(Wxlabel,Wyvalues,width = 0.2)
plt.xlabel("West States")
plt.ylabel("Prediction Accuracy")
plt.title("Logistic Regression Predictions")
plt.show()


#Initial weight vector per feature
import random
def initialWeight(traindatarow):
    weight = []
    numfeatures = len(traindatarow)
    for i in range(numfeatures):
        weight.append(random.uniform(-0.1,0.1))
    return weight


def output(traindata,weight):
    numfeatures = len(traindata)
    output = 1
    for i in range(numfeatures):
        output += traindata[i]*weight[i]

    perceptron = -1
    if (output>0):
        perceptron = 1
    return perceptron




#Update weight
```

```
def updateWeight(weight,traindata,label,rate,iterations):
    o = output(traindata,weight)

    if (label == 0):
        o = output(traindata,weight)
        for j in range(len(traindata)):
            change = rate*traindata[j]*(-1-o)
            weight[j] += change
    else:
        o = output(traindata,weight)
        for j in range(len(traindata)):
            change = rate*traindata[j]*(1-o)
            weight[j] += change
    if (iterations>1):
        weight =
updateWeight(weight,traindata,label,rate,iterations-1)
    return weight


def prediction(traindataTable,weight):
    #Above Average= 1 and Below Average =-1
    features = traindataTable.select('features').collect()
    labels = traindataTable.select('label').collect()
    accuracy = 0

    for i in range(len(features)):
        fvector = features[i][0]
        outcome = labels[i][0]
        sum = output(fvector,weight)

        if (sum==1 and outcome == 1) or (sum == -1 and outcome ==
0):
            accuracy+=1
    return float(accuracy)/len(features)

rate = 0.05
iterations = 6
overallaccuracy = 0
SVMpredictions = []
for table in stateCDF:
    #STATE NAME
    statename = table.select("state").first()[0]

    #Get info to pandas to do mlib
    #Remove attributes that will not be feature or outcome
    newtable =
table.drop("state","year","Deaths","Population","deathpercent")

    #Get all features that can be attributes which is everything
except above average
```

```
    features = []
    for feature in newtable.dtypes:
        if feature[1] != 'int':
            features.append(feature[0])

    #Use vector assembler just to split up data randomly
    vectorass = VectorAssembler(inputCols = features,
outputCol='features')
    vectorDF = vectorass.transform(newtable)
    vectorDF = vectorDF.select(['features', 'label'])

    #Split into training and test data
    train, test = vectorDF.randomSplit([0.7, 0.3],seed = 12)

    featurerows = train.select('features').collect()
    labels = train.select('label').collect()

    #Get predictions
    weight = initialWeight(featurerows[0][0])

    for i in range(len(featurerows)):
        fvector = featurerows[i][0]
        outcome = labels[i][0]

        weight =
updateWeight(weight,fvector,outcome,rate,iterations)

    test_result = prediction(test,weight)
    SVMpredictions.append([statename,test_result])
    print("State: ",statename, " Accuracy: ", test_result)
    overallaccuracy += test_result


#Graph results
import matplotlib.pyplot as plt
SVMxlabel = [prediction[0] for prediction in SVMpredictions]
SVMyvalues = [prediction[1] for prediction in SVMpredictions]

fig = plt.figure(figsize = (30, 6))

# creating the bar plot
plt.bar(SVMxlabel,SVMyvalues,width = 0.2)

plt.xlabel("States")
plt.ylabel("Prediction Accuracy")
plt.title("SVM Predictions")
plt.show()
```

```
# Creating the bar plot for the northern states
SVMNxlabel = [prediction[0] for prediction in SVMpredictions if
prediction[0] in north]
SVMNyvalues = [prediction[1] for prediction in SVMpredictions if
prediction[0] in north]
fig = plt.figure(figsize = (9, 5))
plt.bar(SVMNxlabel,SVMNyvalues,width = 0.2)
plt.xlabel("Northeast States")
plt.ylabel("Prediction Accuracy")
plt.title("SVM Predictions")
plt.show()

# Creating the bar plot for the south states
SVMSxlabel = [prediction[0] for prediction in SVMpredictions if
prediction[0] in south]
SVMSyvalues = [prediction[1] for prediction in SVMpredictions if
prediction[0] in south]
fig = plt.figure(figsize = (9, 5))
plt.bar(SVMSxlabel,SVMSyvalues,width = 0.2)
plt.xlabel("Southern States")
plt.ylabel("Prediction Accuracy")
plt.title("SVM Predictions")
plt.show()

# Creating the bar plot for the Midwest
SVMMxlabel = [prediction[0] for prediction in SVMpredictions if
prediction[0] in midwest]
SVMMyvalues = [prediction[1] for prediction in SVMpredictions if
prediction[0] in midwest]
fig = plt.figure(figsize = (9, 5))
plt.bar(SVMMxlabel,SVMMyvalues,width = 0.2)
plt.xlabel("Midwest States")
plt.ylabel("Prediction Accuracy")
plt.title("SVM Predictions")
plt.show()

# Creating the bar plot for the West
SVMWxlabel = [prediction[0] for prediction in SVMpredictions if
prediction[0] in west]
SVMWyvalues = [prediction[1] for prediction in SVMpredictions if
prediction[0] in west]
fig = plt.figure(figsize = (9, 5))
plt.bar(SVMWxlabel,SVMWyvalues,width = 0.2)
plt.xlabel("West States")
plt.ylabel("Prediction Accuracy")
plt.title("SVM Predictions")
plt.show()
```

```
#Compare and anaylze both results together
import numpy as np
xaxisEntity = np.arange(len(xlabel))
fig = plt.figure(figsize = (30, 6))
plt.bar(xaxisEntity - 0.2, yvalues, 0.3, label = 'Log')
plt.bar(xaxisEntity + 0.2, SVMyvalues, 0.3, label = 'SVM')

plt.xticks(xaxisEntity, xlabel)
plt.xlabel("States")
plt.ylabel("Accuracy Predictions")
plt.title("Comparison of Log vs SVM")
plt.legend()
plt.show()

def getSigmoid(inputList):
    siglist = []
    for x in inputList:
        siglist.append(1 / (1 + math.exp(-x)))
    return np.array(siglist)

def updateWeightBias(featureRows,labels,num_rows,r,weight,bias):
    weightedRows = np.dot(featureRows, weight)+ bias
    sig = getSigmoid(weightedRows)
    sigOutcome = sig-labels.T

    #Change weight and bias values
    weight = weight - ((r/num_rows) *
np.dot(featureRows.T,sigOutcome))
    bias = bias - ((r/num_rows) * np.sum(sigOutcome))
    return weight,bias

def modelLogReg(featureRows,labels,rows,columns,r,iters):
    weight = np.zeros(columns)
    bias = 0
    for i in range(iters):
        weight,bias =
updateWeightBias(featureRows,labels,rows,r,weight,bias)
    return weight,bias

def make_predictions(test,weight,bias):

    testfeaturerows = test.drop('label').toPandas()
    testlabels = test.select('label').toPandas()

    #Normalize features for e overload
    normalized = testfeaturerows.copy()
    for column in testfeaturerows.columns:
        if (column != "label" and
testfeaturerows[column].abs().max() != 0):
```

```
                normalized[column] = testfeaturerows[column]
/testfeaturerows[column].abs().max()

    results = np.dot(normalized, weight)+ bias
    predictions = getSigmoid(results)
    correct = 0
    total = 0
    for prediction in predictions:
        result = 0
        if (prediction > 0.5):
            result = 1
        if (result == testlabels.values[total]):
            correct+=1
        total +=1
    return correct/total

#Log regression
selfLog = []
for table in stateCDF:
    #STATE NAME
    statename = table.select("state").first()[0]

    #Get info to pandas to do mlib
    #Remove attributes that will not be feature or outcome
    newtable =
table.drop("state","year","Deaths","Population","deathpercent")

    #Get all features that can be attributes which is everything
except above average
    features = []
    for feature in newtable.dtypes:
        if feature[1] != 'int':
            features.append(feature[0])
    features.append('label')

    #Use vector assembler just to split up data randomly
    vectorDF = vectorass.transform(newtable)
    vectorDF = vectorDF.select(features)

    #Split into training and test data
    train, test = vectorDF.randomSplit([0.7, 0.3],seed = 12)

    #Get features for the x axis and the label which is the y axis
    featurerows = train.drop('label').toPandas()
    labels = train.select('label').toPandas()

    #Normalize features
    normalized = featurerows.copy()
    for column in features:
```

```
        if (column != "label" and featurerows[column].abs().max()
!= 0):
                normalized[column] = featurerows[column]
/featurerows[column].abs().max()

    weight,bias =
modelLogReg(normalized,labels['label'],train.count(),len(features)-
1,0.05, 6)
    results = make_predictions(test,weight,bias)
    selfLog.append([statename,results])
    print("State: ",statename, " Accuracy: ", results)


#Graph results
xaxisEntity = np.arange(len(xlabel))
fig = plt.figure(figsize = (30, 6))

LOGyvalues = [prediction[1] for prediction in selfLog]

plt.bar(xaxisEntity - 0.2, yvalues, 0.3, label = 'Log')
plt.bar(xaxisEntity + 0.2, LOGyvalues, 0.3, label =
'LogImplemented')

plt.xticks(xaxisEntity, xlabel)
plt.xlabel("States")
plt.ylabel("Accuracy Predictions")
plt.title("Comparison of Pyspark Log vs Log")
plt.legend()
plt.show()
```

VI. REFERENCES

(1) BuzzFeedNews, *Nics-firearm-background-checks: Monthly data from the FBI's National Instant Criminal Background Check System, converted from PDF to CSV.* GitHub (2022) Retrieved May 4, 2022 from https://github.com/BuzzFeedNews/nics-firearm-background-checks

(2) *Gun deaths by county - dataset by Nkrishnaswami*. data.world. (2021, November 6). Retrieved May 4, 2022, from https://data.world/nkrishnaswami/gun-deaths-by-county