AI Models for Auburn Baseball

User's Manual

Authors:

Trent Gavron (tmg0046)

Justin Whisonant (jgw0035)

Austin Beville (acb0157)

Lauren Lyons (lnl0017)

Comp 4710 Senior Design Team #10

Department of Computer Science and Software Engineering Samuel Ginn College of Engineering, Auburn University

Table of Contents

1. System Metaphor

Our system will be used to generate reports for the Auburn baseball team of the optimal defensive positioning for each player in the opposing team's lineup. Our predictive model will display the best defensive positioning by determining which of our pitchers is pitching and how the current batter has performed in previous at-bats against pitchers that have similar metrics (vertical/horizontal break, velocity, spin rate, etc.), similar pitching styles (left vs. right handed, overhand vs. sidearm vs. submarine, etc.), and similar pitch types or arsenals (four-seam fastball, sinker, curveball, slider, etc.). After the filtering process, this information will be displayed on a model of a baseball field, which will show the percentage of ground balls hit to each section of the infield and a heat map of where the flyballs landed in the outfield.

2. Model's Setup

This guide will show users how to run each block of our Jupyter Notebook file and what each block of code does. The database team has been the ones who have worked on and implemented what the end user is actually going to see, so this is more of an overview of how users can see specifically what our model does. The Jupyter Notebook file allows this to be a simple process that could allow users with very minimal knowledge of software engineering to retrieve the results of each of the models.

2.1: Importing Necessary Files and Libraries

```
# Imports
from Models import ModelUtil
from Data import Preprocessing, DataUtil
from Visualization import VisualUtil, batch_image_to_excel
from Logs import logging as logs

import importlib
import configparser
import numpy as np
import pickle

config = configparser.ConfigParser()
config.read('Data//config.ini')

importlib.reload(Preprocessing)
importlib.reload(ModelUtil)
importlib.reload(VisualUtil)
importlib.reload(batch_image_to_excel)
importlib.reload(logs)

import pandas as pd
from sklearn.preprocessing import MinMaxScaler

import warnings
warnings.filterwarnings("ignore")
```

```
# 1) Load all data from preprocessing
importlib.reload(Preprocessing)
newprocessing = 'True' in config['DATA']['USE_NEW_PREPROCESSING']
infieldDataFrame, outfieldDataFrame = Preprocessing.dataFiltering([], newprocessing)
```

The import section of the notebook ensures that all necessary files and libraries have been imported into the program. You **always** need to start the session by running this code block, otherwise the rest of the notebook will give errors. This block also pre-processes the data based on the values set in the config file, so if you want to pull data from a different source or need to quickly debug something, you can change the config appropriately and then re-run this code block.

2.2: Correlation Matrix

```
infieldDF4Matrix = infieldDataFrame.copy()
outfieldDF4Matrix = outfieldDataFrame.copy()
strColumns = []
for cName in outfieldDF4Matrix.columns:
    if(str(outfieldDF4Matrix[cName].dtype) in 'object'):
        strColumns.append(cName)
rValueDict = {}
for cName in strColumns:
    i = 0
    infieldUniques = infieldDF4Matrix[cName].unique()
    for rValue in infieldUniques:
        rValueDict.update({rValue:i})
        i+=1
    infieldDF4Matrix[cName] = infieldDF4Matrix[cName].map(rValueDict)
    uniqueVals = [x for x in outfieldDF4Matrix[cName].unique() if x not in infieldUniques]
    for rValue in uniqueVals:
        rValueDict.update({rValue:i})
        i+=1
    outfieldDF4Matrix[cName] = outfieldDF4Matrix[cName].map(rValueDict)
infieldDF4Matrix = infieldDF4Matrix.replace(np.nan, 0)
infieldDF4Matrix = infieldDF4Matrix.replace('', 0)
outfieldDF4Matrix = outfieldDF4Matrix.replace(np.nan, 0)
outfieldDF4Matrix = outfieldDF4Matrix.replace('', 0)

# Correlation does not imply causation.
# -1 means that the 2 variables have an inverse linear relationship: when X increases, Y decreases
# 0 means no linear correlation between X and Y
# 1 means that the 2 variables have a linear relationship: when X increases, Y increases too.
infieldcorrmatrix = infieldDF4Matrix.corr()
outfieldcorrmatrix = outfieldDF4Matrix.corr()
if (config['LOGGING']['Excel'] == 'True'):
    logs.writeToExcelSheet(infieldcorrmatrix, "Infield Correlation Matrix")
    logs.writeToExcelSheet(outfieldcorrmatrix, "Outfield Correlation Matrix")
if (config['LOGGING']['Debug'] == 'True'):
    print(infieldcorrmatrix)
    print(outfieldcorrmatrix)
```

This section outputs a correlation matrix for visualizing the relationship between different features in the dataset.

2.3: Running the Models

```
importlib.reload(logs)
# 2) Trains all Models and exports all data to an Excel Sheet
max_depth = 50
max_features = 30
max_leaf_nodes = 150
# could also add ways to change it for these hyperparams below for other models
var_smoothing = 1e-9
lr = 0.8
e = 100
rC = 1
kernel='linear'
degree= 1
gamma= 'scale'
coef0= 0.0

runCount = int(config['TRAIN']['TimesRun'])
if ("False" in config['TRAIN']['Testing']):
    runCount = 1
    print("Not Testing")
for j in range(runCount):
        xTrain, xTest, yTrain, yTest = ModelUtil.modelDataSplitting(infieldDataFrame, j, 0.25,'InfieldTrainingFilter')
        if("True" in config['MODELS']['DTC']):
            dtOutput = ModelUtil.runDT(xTrain, yTrain, xTest, yTest, max_depth, max_features, max_leaf_nodes)
            if ("True" in config['DATA']['Pickle']):
                # Save the model to a file
                with open('Models/DecisionTree.pkl', 'wb') as file:
                    pickle.dump(dtOutput, file)
        if("True" in config['MODELS']['NB']):
            nbOutput = ModelUtil.runNB(xTrain, yTrain, xTest, yTest, var_smoothing)
            if ("True" in config['DATA']['Pickle']):
                # Save the model to a file
                with open('Models/NaiveBayes.pkl', 'wb') as file:
                    pickle.dump(nbOutput, file)
        if("True" in config['MODELS']['LR']):
            logRegOutput = ModelUtil.runLogReg(xTrain, yTrain, xTest, yTest, lr, e)
            if ("True" in config['DATA']['Pickle']):
                # Save the model to a file
                with open('Models/LogRegression.pkl', 'wb') as file:
                    pickle.dump(logRegOutput, file)
        if("True" in config['MODELS']['SVM']):
            svmOutput = ModelUtil.runSVM(xTrain, yTrain, xTest, yTest, rC, kernel, degree, gamma, coef0)
            if ("True" in config['DATA']['Pickle']):
                # Save the model to a file
                with open('Models/SVM.pkl', 'wb') as file:
                    pickle.dump(svmOutput, file)
        # if("True" in config['MODELS']['RF']):
        #     for i in range(0, len(trainIn)):
        #         direction, distance = ModelUtil.runRFR(trainIn[i], trainOut[i], testIn[i], testOut[i])
```

This section trains all of the models based on the config settings. If pickle is set to true, then it will also save the model as a pickle compressed file (to be loaded later).

```
importlib.reload(ModelUtil)
importlib.reload(logs)
# a) Decision Tree
# Need to test these hyperparameters for best case
# Maybe make a way to superset these
max_depth =      [50, 40]
max_features =   [30, 20]
max_leaf_nodes = [150, 100]
hyperparamlist = []
# This just makes the permutations of the hyperparameters above. Lets you test on many hyperparams.
for n in range(len(max_depth)):
    for k in range(len(max_features)):
        for m in range(len(max_leaf_nodes)):
            hyperparamlist.append([max_depth[n], max_features[k], max_leaf_nodes[m]])

# for each permutation, it runs a certain amount of time that you specify in the config (30 rn bc of Dozier)
# and saves the outcome to an excel sheet
# requires to rerun the training set every time because otherwise will give you the same outcome every time
# Also proves that its the models ability, not the luck of the draw for the data
for lst in hyperparamlist:
    runCount = int(config['TRAIN']['TimesRun'])
    if ("False" in config['TRAIN']['Testing']):
        runCount = 1
        print("Not Testing")
    for j in range(runCount):
        xTrain, xTest, yTrain, yTest = ModelUtil.modelDataSplitting(infieldDataFrame, j, 0.25,'InfieldTrainingFilter')
        dtOutput = ModelUtil.runDT(xTrain, yTrain, xTest, yTest, lst[0], lst[1], lst[2])
        if ("True" in config['DATA']['Pickle']):
            # Save the model to a file
            with open('Models/DecisionTree.pkl', 'wb') as file:
                pickle.dump(dtOutput, file)
```

This section runs the decision tree model alone, also based on the settings found in the config file.

```
importlib.reload(ModelUtil)
importlib.reload(logs)
# b) Naive Bayes

var_smoothing = 1e-9
runCount = int(config['TRAIN']['TimesRun'])
if ("False" in config['TRAIN']['Testing']):
    runCount = 1
    print("Not Testing")
for j in range(runCount):
        xTrain, xTest, yTrain, yTest = ModelUtil.modelDataSplitting(infieldDataFrame, j, 0.25,'InfieldTrainingFilter')
        nbOutput = ModelUtil.runNB(xTrain, yTrain, xTest, yTest, var_smoothing)
```

This section runs the naive bayes model training and output.

```
importlib.reload(ModelUtil)
importlib.reload(logs)
# c)Logistic Regression
lr = 0.8
e = 100
runCount = int(config['TRAIN']['TimesRun'])
if ("False" in config['TRAIN']['Testing']):
    runCount = 1
    print("Not Testing")
for j in range(runCount):
    xTrain, xTest, yTrain, yTest = ModelUtil.modelDataSplitting(infieldDataFrame, j, 0.25,'InfieldTrainingFilter')
    logRegOutput = ModelUtil.runLogReg(xTrain, yTrain, xTest, yTest, lr, e)
```

This section runs the logistic regression model.

```
importlib.reload(ModelUtil)
importlib.reload(logs)
# d) SVM
rC = 1
kernel='linear'
degree= 1
gamma= 'scale'
coef0= 0.0
runCount = int(config['TRAIN']['TimesRun'])
if ("False" in config['TRAIN']['Testing']):
    runCount = 1
    print("Not Testing")
for j in range(runCount):
    xTrain, xTest, yTrain, yTest = ModelUtil.modelDataSplitting(infieldDataFrame, j, 0.25,'InfieldTrainingFilter')
    svmOutput = ModelUtil.runSVM(xTrain, yTrain, xTest, yTest, rC, kernel, degree, gamma, coef0)
```

This section runs a support vector machine model (not used currently due to only predicting as field slice 2 or 4)

2.4: Logging the Results

```
if("True" in config['LOGGING']['Excel']):
    sColumns = ['Training Accuracy', 'Testing Accuracy', 'Training Average Error', 'Testing Average Error',
                'Training F1(micro)', 'Training F1(macro)', 'Training F1(weighted)', 'Testing F1(micro)',
                'Testing F1(macro)', 'Testing F1(weighted)', 'Training AUC(macro)', 'Training AUC(weighted)',
                'Testing AUC(macro)', 'Testing AUC(weighted)', 'Section 0 Probability', 'Section 1 Probability',
                'Section 2 Probability', 'Section 3 Probability', 'Section 4 Probability']
    if("True" in config['MODELS']['DTC']):
        # columns in excel: I J K L W X Y Z AA AB AC AD AE AF AG AH AI AJ AK
        sColumnsLetter = ['I','J','K','L','W','X','Y','Z','AA','AB','AC','AD','AE','AF','AG','AH','AI','AJ','AK']
        logs.excelAverages('DecisionTree',sColumns,sColumnsLetter)
    if("True" in config['MODELS']['NB']):
        sColumnsLetter = ['D','E','F','G','R','S','T','U','V','W','X','Y','Z','AA','AB','AC','AD','AE','AF']
        logs.excelAverages('NaiveBayes',sColumns,sColumnsLetter)
    if("True" in config['MODELS']['LR']):
        sColumnsLetter = ['E','F','G','H','S','T','U','V','W','X','Y','Z','AA','AB','AC','AD','AE','AF','AG']
        logs.excelAverages('LogisticRegression',sColumns,sColumnsLetter)
    if("True" in config['MODELS']['SVM']):
        sColumnsLetter = ['H','I','J','K','V','W','X','Y','Z','AA','AB','AC','AD','AE','AF','AG','AH','AI','AJ']
        logs.excelAverages('SVM',sColumns,sColumnsLetter)
    if("True" in config['MODELS']['RF']):
        logs.excelAverages('RandomForest',sColumns,sColumnsLetter)
```

This section logs the results to an excel file.

2.5: Testing Individual Data Points

```
# Change the value of index to look at different datapoints
importlib.reload(VisualUtil)
# 3) Model Testing:
dt = dtOutput[0]
nb = nbOutput[0]
logReg = logRegOutput[0]
# svm = svmOutput[0]

print("Testing Output: ")
# index of test value:
index = 4555
print(f"Actual Field Slice: \t\t{yTest.iloc[index]}")

print("\nDecision Tree:")
print(f"Predicted Field Slice: \t\t{dt.predict([xTest.iloc[index]])[0]}")
print(f"Field Slice Probabilities: \t{dt.predict_proba([xTest.iloc[index]])[0]}")

print("\nNaive Bayes:")
print(f"Predicted Field Slice: \t\t{nb.predict([xTest.iloc[index]])[0]}")
print(f"Field Slice Probabilities: \t{nb.predict_proba([xTest.iloc[index]])[0]}")

print("\nLogistic Regression:")
print(f"Predicted Field Slice: \t\t{logReg.predict([xTest.iloc[index]])[0]}")
print(f"Field Slice Probabilities: \t{logReg.predict_proba([xTest.iloc[index]])[0]}")

# print("\nSVM:")
# print(f"Predicted Field Slice: \t\t{svm.predict([xTest.iloc[index]])[0]}")
# print(f"Field Slice Probabilities: \t{svm.predict_proba([xTest.iloc[index]])[0]}")

averageProbs = dt.predict_proba([xTest.iloc[index]])[0] + nb.predict_proba([xTest.iloc[index]])[0] \
    + logReg.predict_proba([xTest.iloc[index]])[0] # + svm.predict_proba([xTest.iloc[index]])[0]
averageProbs = averageProbs / 3

print(f"\n\nAVG Prediction: \t\t{np.argmax(averageProbs)+1}")
print(f"Field Slice AVG Probabilities: \t{averageProbs}")

VisualUtil.visualizeData(averageProbs, [1], 'TestPic.png')
```

This section can be used to test individual data points in the test set (change the value of index to change the datapoint). This is helpful for ensuring that the results make sense and are predicting close to the actual value of where the ball was hit.

2.6: Data Visualization

```
# 5) Data Visualization
importlib.reload(VisualUtil)

# Temporary method of getting percentages for testing purposes
infieldPercentages  = np.random.dirichlet(np.ones(4), size=1)[0]
outfieldPercentages = np.random.dirichlet(np.ones(2), size=1)[0]
outfieldCoordinates = np.random.uniform(low=[-45, 150], high=[45, 400], size=(30,2))

VisualUtil.visualizeData(infieldPercentages, outfieldCoordinates, "FieldTest")
```

This block demonstrates how to call the visualization function from VisualUtil. The inputs are the infield percentages from the model, the outfield percentages or the outfield coordinates (depending on if you want the see slices in the outfield or a heatmap in the outfield), and the name of the output image. In this example, the data being generated is 30 uniformly random data points with a direction between -45 and 45 degrees. The outfield coordinates generate an extra parameter, the distance, of values between 150 and 400 ft (about the range of the grassline to the fenceline).

2.7: Factoring in Pitcher's Metrics

```
# Average Pitcher Data Processing and Running
importlib.reload(Preprocessing)
importlib.reload(DataUtil)
importlib.reload(VisualUtil)
importlib.reload(batch_image_to_excel)

pitchingAveragesDF = DataUtil.getRawDataFrame('Data/PitchMetricAverages_AsOf_2024-03-11.csv')
# drop nan values from the used columns
specific_columns = ["PitcherThrows", "BatterSide", "TaggedPitchType", "RelSpeed",
                    "InducedVertBreak", "HorzBreak", "RelHeight", "RelSide", "SpinAxis",
                    "SpinRate", "VertApprAngle", "HorzApprAngle"] # pitcher averages
infieldDataFrame = infieldDataFrame[specific_columns]
averagesX = pitchingAveragesDF[specific_columns] # pitcher averages
#averagesX = averagesX[["PitcherThrows", "BatterSide", "TaggedPitchType", "PlateLocHeight",
#"PlateLocSide", "ZoneSpeed", "RelSpeed", "SpinRate", "HorzBreak", "VertBreak"]]

averagesX["PitcherThrows"] = averagesX["PitcherThrows"].map({"Left":1, "Right":2, "Both":3})
averagesX["BatterSide"] = averagesX["BatterSide"].map({"Left":1, "Right":2})
averagesX["TaggedPitchType"] = averagesX["TaggedPitchType"].map({"Fastball": 1, "FourSeamFastBall":1,
                                                                 "Sinker":2, "TwoSeamFastBall":2,
                                                                 "Cutter":3, "Curveball":4,
                                                                 "Slider":5, "ChangeUp":6,
                                                                 "Splitter":7, "Knuckleball":8})
# normalize this based on min and maxes from training data
averagesX = DataUtil.normalizeData(averagesX, infieldDataFrame)

# Change the value of index to look at different datapoints
importlib.reload(VisualUtil)
# 3) Model Testing:
dt = dtOutput[0]
nb = nbOutput[0]
logReg = logRegOutput[0]
# svm = svmOutput[0]
for index in range(pitchingAveragesDF.shape[0]):
    print(index)
    averageProbs= []
    averageProbs = dt.predict_proba([averagesX.iloc[index]])[0] + nb.predict_proba([averagesX.iloc[index]])[0] \
        + logReg.predict_proba([averagesX.iloc[index]])[0]
    averageProbs = averageProbs / 3

    # print(f"\n\nAVG Prediction: \t\t{np.argmax(averageProbs)+1}")
    # print(f"Field Slice AVG Probabilities: \t{averageProbs}")
    fileName = pitchingAveragesDF.iloc[index][0].replace(",", "_").replace(" ", "") + "_" \
        + pitchingAveragesDF.iloc[index]["TaggedPitchType"] + "_" + pitchingAveragesDF.iloc[index]["BatterSide"] \
            + "Batter"
    VisualUtil.visualizeData(averageProbs, [1], fileName)

batch_image_to_excel.create_excel()
```

This section uses the pitchers average dataset which we obtained as a csv file and predicts those data points based on the models which are trained above.