

AI Models for Auburn Baseball

Developer'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

Overview	3
Installation	4
Important Files	5
System Design	7
Data	8
Logging	10
Models	12
Output	17
Visualization	18

Overview

The Auburn Baseball AI program aims to provide the baseball coaching staff a useful, transparent, and consistent predictive AI model that can provide accurate forecasts of where the opposing team will hit the ball. Based on this information, the coaches can elect to use what the model says or their own judgment in order to shift the defense's position on the field. The program takes advantage of the fact that it can use all of the statistics of a pitcher as well as those of the opposing batter to provide unique individual matchups on a pitch-to-pitch basis between a specific pitcher and a specific batter. Once the program has processed the data through the predictive models, it outputs the data through a visualization of the baseball field, showcasing the different scenarios that it believes will play out as shaded probabilities across different slices of the infield and as a heatmap throughout the outfield.

A major factor in the way the codebase was designed was quick feedback, modularity, transparency, and modifiability. Each component of the program can be easily tweaked and changed to meet the needs of the coaches. The models are continuously integrating new data on a weekly basis thanks to the database team, which further helps keep the models relevant and accurate. The program uses widely known machine learning modules such as SciKitLearn as well as popular data handling modules like Pandas in order to keep onboarding simple and progress efficient. The interface with the Auburn University site Data-Getta is the most recent addition as of writing this, and the new team will likely need to also become familiar with how it operates since this is where the end-user will see the visualizations and data outputs.

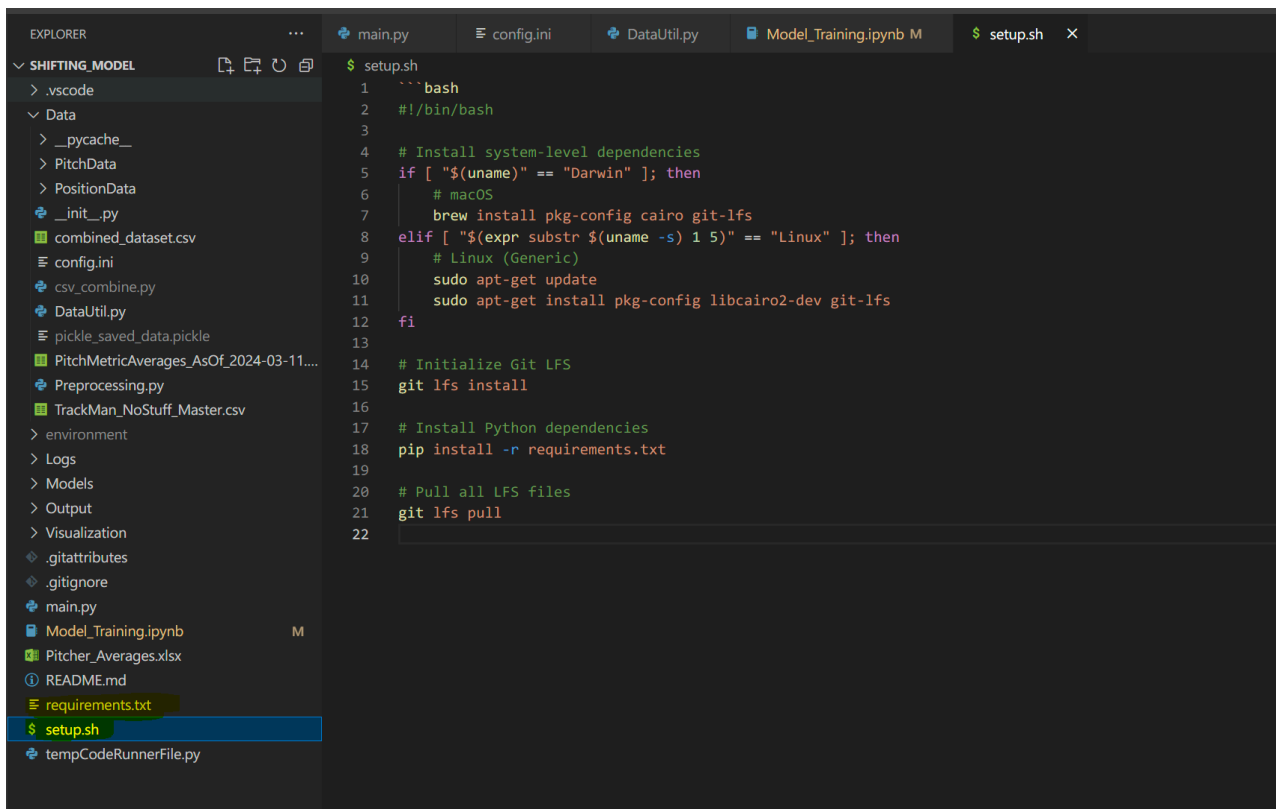
This document aims to serve as a guide for those who will be working on this program and hopefully provide enough information for them to fill the roles of the previous team. It will explain how to install the necessary dependencies, showcase the various files and scripts housed in the repository, and explain each function and how they can be used. If you have a question that was not answered within this document, feel free to reach out to us using our emails on the cover page.

Installation

Once the github repo has been cloned, either run the setup.sh script to install all requirements, or call:

“pip install -r requirements.txt”

If installing using pip, you will also need to manually install cairo and git-lfs (the commands are found in setup.sh) as well as setup git lfs to store and load large files such as our training csv.



The screenshot shows a VS Code editor with a dark theme. The Explorer sidebar on the left lists the project files, with `requirements.txt` and `setup.sh` highlighted. The main editor area displays the contents of `setup.sh`, which is a shell script for installing dependencies on macOS and Linux.

```
$ setup.sh
1  ``bash
2  #!/bin/bash
3
4  # Install system-level dependencies
5  if [ "$(uname)" == "Darwin" ]; then
6      # macOS
7      brew install pkg-config cairo git-lfs
8  elif [ "$(expr substr $(uname -s) 1 5)" == "Linux" ]; then
9      # Linux (Generic)
10     sudo apt-get update
11     sudo apt-get install pkg-config libcairo2-dev git-lfs
12 fi
13
14 # Initialize Git LFS
15 git lfs install
16
17 # Install Python dependencies
18 pip install -r requirements.txt
19
20 # Pull all LFS files
21 git lfs pull
22
```

Important Files

Data/config.ini:

- Here is where you will find our config settings which can be adjusted to change the operation of the models
- We have options to enable/disable logging, debugging, specific models, as well as many options for what data is being used to train the models
- These settings are used in both the main.py and Model_Training.ipynb files, so they must be configured before using those files (including when uploading to the database server)

Model_Training.ipynb:

- This file is our main testing file where you will find many blocks of code which can be run individually
- This is where we first set our models up and it is the best place to test changes due to being able to run certain blocks while storing outputs from others
 - This means we can run our data preprocessing, then make tweaks to our models or other functions without having to re-run our preprocessing (this saves a lot of time)
- This file calls methods and functions that are found all throughout our codebase (and are described below)
 - Hitting f12 with a method or function highlighted (Visual Studio Code) will take you to the declaration of that definition (even within other files)
- Use this file to test new additions and changes to the models, or any supporting functions before transferring those changes to main.py (if needed)

Main.py:

- This file is our main driver file which is run weekly on the database server
- This file is set up to retrain our models, then re-predict all pitcher averages based on newly updated inputs
 - Our models still currently only train from a combined csv with historical data as

the database team has not added data from years past yet

- Once they have, there are functions partially set up to use that data for training
- This file does use methods from elsewhere in our codebase as well
- The main functions of this file currently:
 - Train the models
 - Connect to the database and load pitcher averages
 - Predict using our models with those averages
 - Output and write those averages to the database
 - From there they are used to display our outputs on each pitcher's defensive shift page

[illegible]

Data

DataUtil.py:

`getData()`

- Input: None directly; uses global configuration settings to determine the data source.
- Output: DataFrame containing data from various sources depending on configuration settings (database, FTP, local files).

`databaseConnect()`

- Input: None directly; uses a hardcoded database URL.
- Output: A cursor and connection object for the PostgreSQL database.

`getPitcherAverages(cur, infieldDataFrame, outfieldDataFrame, teamFilter)`

- Input: Database cursor, infield data frame, outfield data frame, team filter criteria.
- Output: Normalized pitcher averages and a raw data frame containing metadata.

`writePitcherAverages(cur, conn, key, values)`

- Input: Database cursor, connection, keys (identifiers), and values (model values to be inserted).
- Output: Updates or inserts pitcher averages into a database. No direct data output; performs database operations.

`getDBPitchData()`

- Input: None directly; connects to a database to retrieve data.
- Output: DataFrame containing pitch data joined with batter data from a database.

`getFTPData()`

- Input: None directly; configuration settings for FTP connection and file retrieval.
- Output: DataFrame containing pitch data from FTP files within specified date ranges.

`getRawDataFrame(filename, rows)`

- Input: Filename or path to a CSV file and rows to include (if already fetched).
- Output: A pandas DataFrame with selected columns based on global settings.

`convertSQLToList(data)`

- Input: Data fetched from SQL queries.
- Output: List of lists where each sublist is a row from the SQL data, with decimal values converted to strings.

saveDataToPickle(filename, pickle_file_path)

- Input: Filename to read from and pickle file path to save to.
- Output: None; serializes data to a pickle file.

getPickleData(pickle_file_path)

- Input: Path to a pickle file containing serialized data.
- Output: DataFrame created from the deserialized data

Preprocessing.py:

dataProcessing()

- Description: This function orchestrates the full pipeline of reading data, cleaning it, converting categorical data to numerical format, and normalizing it for further use.
- Input: No explicit inputs; implicitly depends on DataUtil for data.
- Output: A DataFrame (normalizedDataFrame) that has been normalized after cleaning and converting categorical variables to numerical format.

dataFiltering(df, useNewProcessing)

- Description: Filters data based on a specified method (new or old processing), splitting data into infield and outfield datasets based on certain criteria.
- Input:
 - df: DataFrame containing data to be filtered.
 - useNewProcessing: Boolean that determines which filtering logic to apply.
- Output: Tuple of DataFrames for infield and outfield data, along with feature headers if new processing is used. Returns infield and outfield DataFrames directly if old processing is used.

Combined_dataset.csv:

This is our largest training set which comprises of historical data, should be replaced with full database training once enough data points are added

Logging

`writeLog(log, name="", descriptor=".txt")`

- Description: Writes log entries to a text file with a timestamp in its name.
- Input:
 - log: List of strings (log entries).
 - name: Optional base name for the log file.
 - descriptor: File extension, defaults to ".txt".
- Output: None. Saves a log file to disk.

`logModel(...)`

- Description: Constructs a detailed log for a machine learning model's performance and configurations, then calls `writeLog` to save it.
- Input: Parameters include model type, model object, training and testing statistics, data partitions, model parameters, and a descriptor of whether it's an infield or outfield model.
- Output: A comprehensive log file is written with performance statistics and model details.

`printModel(...)`

- Description: Similar to `logModel` but prints the model's details and performance statistics directly to the console.
- Input: Same as `logModel`.
- Output: Direct console outputs of the model performance details.

`writeToExcelSheet(logDF, name="")`

- Description: Writes a DataFrame to an Excel sheet. If the file or sheet doesn't exist, it creates them.
- Input:
 - logDF: DataFrame to be written.
 - name: Sheet name in the Excel file.
- Output: An Excel file with the data saved under the specified sheet name.

`writeToImageExcelSheet(picList, name="")`

- Description: Writes images along with captions into an Excel sheet, adjusting their size for the Excel format.

- Input:
 - picList: List of image file paths.
 - name: Excel sheet name.
- Output: An Excel file where images are inserted into the specified sheet.

ExcelModel(...)

- Description: Prepares a DataFrame of model statistics and writes it to an Excel sheet using writeToExcelSheet.
- Input: Extensive model-related data including type, performance stats, parameters, and predictions.
- Output: Writes a DataFrame to an Excel sheet for visualization and record-keeping.

colsum(arr, n, m)

- Description: Calculates the column sums for a 2D array, normalizing by the number of rows.
- Input:
 - arr: 2D list or array.
 - n: Number of columns.
 - m: Number of rows.
- Output: A list containing the average of each column.

excelAverages(modelType, sColumns, sColumnsLetter)

- Description: Averages selected statistical metrics across all runs of a model type and writes them to an Excel sheet.
- Input:
 - modelType: Model type identifier.
 - sColumns: Columns to average.
 - sColumnsLetter: Excel column letters corresponding to sColumns.
- Output: Writes averaged statistics to an Excel workbook.

Models

.pkl files:

InfieldDecisionTree.pkl

- This is a file compressed by pickle-ing it. It is a pkl of the trained Infield Decision Tree Classifier Model. This is so that the end program can train these models, store them, and then reload them to make predictions when requested.

InfieldLogRegression.pkl

- This is a file compressed by pickle-ing it. It is a pkl of the trained Infield Logistic Regression Classifier Model. This is so that the end program can train these models, store them, and then reload them to make predictions when requested.

InfieldNaiveBayes.pkl

- This is a file compressed by pickle-ing it. It is a pkl of the trained Infield Naive Bayes Classifier Model. This is so that the end program can train these models, store them, and then reload them to make predictions when requested.

OutfieldDecisionTree.pkl

- This is a file compressed by pickle-ing it. It is a pkl of the trained Outfield Decision Tree Classifier Model. This is so that the end program can train these models, store them, and then reload them to make predictions when requested.

OutfieldLogRegression.pkl

- This is a file compressed by pickle-ing it. It is a pkl of the trained Outfield Logistic Regression Classifier Model. This is so that the end program can train these models, store them, and then reload them to make predictions when requested.

OutfieldNaiveBayes.pkl

- This is a file compressed by pickle-ing it. It is a pkl of the trained Outfield Naive Bayes Classifier Model. This is so that the end program can train these models, store them, and then reload them to make predictions when requested.

ModelUtil.py:

trainHyperParameters(model, grid, train_x, train_y)

- Description: This is supposed to get the best set of hyperparameters for the different models.
- Input: model is the instance of the machine learning model. Train_x and train_y are the dataframes that host the datapoints that are put into the model.
- Output:

`runRFR(train_x, train_y, test_x, test_y)`

- Description: This is the Random Forest Regressor model.
- Input: train_x is the Dataframe that holds all the features other than the label. train_y is the Dataframe that holds the label for train_x. The entire dataset is split in between training and testing. So, the test_x and test_y are the exact same, but are just different data points.
- Output: This returns the directionScore and distanceScore.

`runDT(train_x, train_y, test_x, test_y, max_depth, max_features, max_leaf_nodes, fieldModelType)`

- Description: This is the Decision Tree Classifier Model. It can run for both Infield and Outfield models as long as you have specified that in the parameter fieldModelType.
- Input: train_x is the Dataframe that holds all the features other than the label. train_y is the Dataframe that holds the label for train_x. The entire dataset is split in between training and testing. So, the test_x and test_y are the exact same, but are just different data points. max_depth, max_features, max_leaf_nodes are all different hyperparameters.
- Output: This returns the DecisionTreeClassifier Instance, trainStats, and testStats. trainStats and testStats holds statistics about the two dataframes that hold training and testing data.

`runNB(train_x, train_y, test_x, test_y, var_smoothing, fieldModelType)`

- Description: This is the Naive Bayes Classifier Model. It can run for both Infield and Outfield models as long as you have specified that in the parameter fieldModelType.
- Input: train_x is the Dataframe that holds all the features other than the label. train_y is the Dataframe that holds the label for train_x. The entire dataset is split in between training and testing. So, the test_x and test_y are the exact same, but are just different data points. var_smoothing is a hyperparameter.

- Output: This returns the NaiveBayesClassifier Instance, trainStats, and testStats. trainStats and testStats holds statistics about the two dataframes that hold training and testing data.

runLogReg(train_x, train_y, test_x, test_y, lr, e, fieldModelType)

- Description: This is the Logistic Regression Classifier Model. It can run for both Infield and Outfield models as long as you have specified that in the parameter fieldModelType.
- Input: train_x is the Dataframe that holds all the features other than the label. train_y is the Dataframe that holds the label for train_x. The entire dataset is split in between training and testing. So, the test_x and test_y are the exact same, but are just different data points. lr and e are two hyperparameters.
- Output: This returns the LogisticRegressionClassifier Instance, trainStats, and testStats. trainStats and testStats holds statistics about the two dataframes that hold training and testing data.

runSVM(train_x, train_y, test_x, test_y, rC, kernel, degree, gamma, coef0, fieldModelType)

- Description: This is the Support Vector Machine Classifier Model. It can run for both Infield and Outfield models as long as you have specified that in the parameter fieldModelType
- Input: train_x is the Dataframe that holds all the features other than the label. train_y is the Dataframe that holds the label for train_x. The entire dataset is split in between training and testing. So, the test_x and test_y are the exact same, but are just different data points. rC, kernel, degree, gamma, and coef0 are hyperparameters.
- Output: This returns the SupportVectorMachineClassifier Instance, trainStats, and testStats. trainStats and testStats holds statistics about the two dataframes that hold training and testing data.

measurePerformance(predictions, test_y)

- Description: Evaluates the performance of a prediction model by comparing its predictions to the actual results.
- Input:
 - predictions: an array containing the predicted values.
 - test_y: an array containing the actual values.

- Output: a measure of the model's performance, such as accuracy or another statistical measure.

`calculateScore(y_true, y_pred)`

- Description: Computes a score indicating the accuracy or effectiveness of predictions compared to the true values.
- Input:
 - `y_true`: an array of true (actual) values.
 - `y_pred`: an array of predicted values by the model.
- Output: a numeric score representing the model's prediction accuracy.

`convertAngleToSlice(angle)`

- Description: Converts an angle into a slice number based on predefined angle ranges.
- Input:
 - `angle`: a numeric value representing an angle.
- Output: an integer representing the slice number corresponding to the given angle.

`colsum(arr, n, m)`

- Description: Calculates the sum of each column in a given 2D array or matrix.
- Input:
 - `arr`: a 2D list or numpy array.
 - `n`: the number of rows in the array.
 - `m`: the number of columns in the array.
- Output: a list or array containing the sum of each column.

`get_infield_statistics(pred, y_test, probs)`

- Description: Computes various statistics for infield predictions including accuracy and other relevant metrics.
- Input:
 - `pred`: an array of predicted infield outcomes.
 - `y_test`: an array of actual infield outcomes.
 - `probs`: an array of probabilities associated with each prediction.
- Output: a dictionary containing computed infield statistics.

`get_outfield_statistics(pred, y_test, probs)`

- Description: Computes various statistics for outfield predictions such as accuracy and other relevant metrics.
- Input:
 - pred: an array of predicted outfield outcomes.
 - y_test: an array of actual outfield outcomes.
 - probs: an array of probabilities associated with each prediction.
- Output: a dictionary containing computed outfield statistics.

`modelDataSplitting(dF, randomState, testSize, dFType, fieldModelType)`

- Description: Splits the provided dataset into training and testing sets according to specified parameters.
- Input:
 - dF: the DataFrame to be split.
 - randomState: an integer that is the seed for the random number generator.
 - testSize: a float representing the proportion of the dataset to include in the test split.
 - dFType: the type of data frame, e.g., infield or outfield data.
 - fieldModelType: the type of model, e.g., predictive or analytical.
- Output: two DataFrames representing the training and testing datasets respectively.

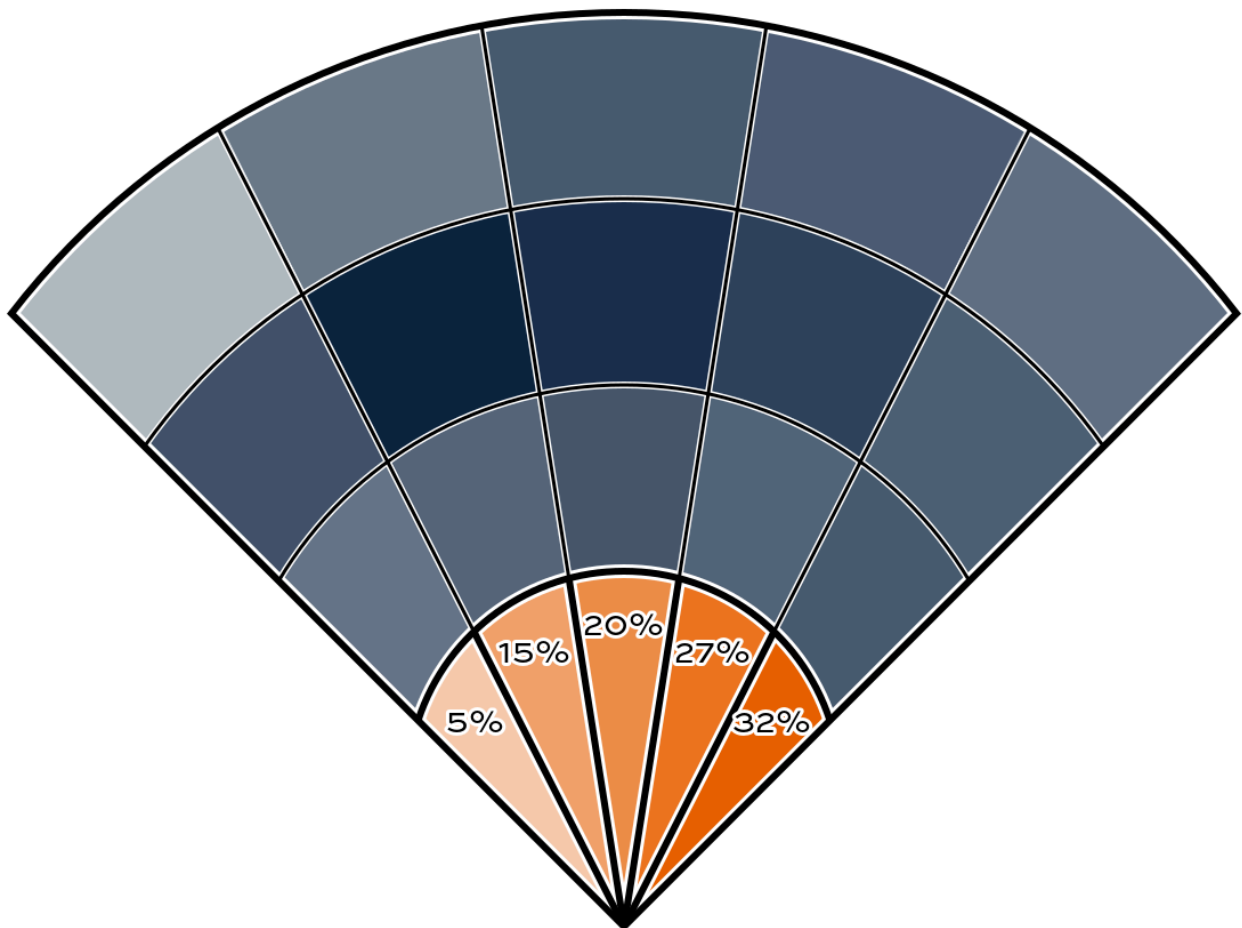
Output

This folder holds all of the different output images that our program can create. Currently they are split up by Pitcher, Pitch-Type, and which hand the batter uses. These images will be put into the Excel Worksheet for our program that shows all of our Pitchers. In collaboration with the Data Getta application, that application makes the images inside of their program.

An example image for the Output folder is:

Chase Allsup - Slider - Left-handed batter

Allsup_Chase_Slider_LeftBatter.png



Visualization

visualizeData(infieldStats, outfieldStats, filename):

This function generates the data visualization as either slices or a heatmap depending on the input parameters and config settings.

initializeFieldVariables():

This function initializes global variables related to the dimensions and coordinates of the baseball field.

drawField(draw, infield, outfield, thick, color):

Draws the lines and slices of the baseball field, distinguishing between infield and outfield.

drawOnlyInfield(draw, infield):

Draws only the infield portion of the baseball field.

drawFieldLines(draw, thick, color):

Draws the baselines and the fenceline of the baseball field.

drawFieldSplit(draw, distance, thick, color):

Draws a line which splits the field across its width. Used for the split between infield and outfield as well as for the heatmap cells.

drawInfieldSliceLines(draw, thick, color, slices):

Draws lines representing slices in the infield portion of the field.

drawOutfieldSliceLines(draw, thick, color, slices):

Draws lines representing slices in the outfield portion of the field.

drawSliceLine(draw, start, end, thick, color):

Draws a line segment between two points.

fillSlices(draw, slices, percentages, arc_distance, arc_angle, color1, color2):

Fills the slices of the field with colors based on provided percentages.

drawFilledSlice(draw, angle_from, angle_to, color, radius):

Draws a filled slice of the field.

drawText(image, text, position):

Draws text on an image.

drawHeatmapLattice(draw, vert_density, horz_density, thick, color):

Draws lattice lines for the heatmap.

drawVertLattice(draw, density, thick, color):

Draws vertical lattice lines.

drawHorzLattice(draw, density, thick, color):

Draws horizontal lattice lines.

fillHeatmap(draw, heatmap, color1, color2):

Fills the heatmap with colors based on provided data.

debugPlot(draw, stats, color):

Plots debug points on the field. Used to ensure heatmap is correct.

drawPoint(draw, bearing, distance, color):

Draws a point on the field.

addPercents(image, slices, percentages, arc_distance):

Adds percentage text to each slice.

blendColors(color1, color2, ratio):

Blends two colors based on a ratio. Color1 being the low value color and Color2 being the high value color. The higher the ratio, the closer to Color2 the blend is.

cleanNumber(num):

Cleans and formats a number as a percentage.

convertToMoundSpace(stats, vert_density, horz_density):

Converts statistics data into heatmap space.

getIntersection(circle, line_start, angle):

Calculates the intersection point of a line with a circle. Used to find where slices intersect with the grassline and the fenceline.

getArc(radius):

Calculates the arc angle based on the radius. The returned value is the angle from the centerline to the intersection of the line that runs from the mound to the baseline at the specified radius.

quadratic(a, b, c):

Solves a quadratic equation.

flip_y(coords):

Flips the y-coordinate. Useful for converting from realspace to picturespace.

flip_x(coords):

Flips the x-coordinate.

flip_xy(coords):

Flips both x and y coordinates.

color10to255(color):

Converts color values from a scale of 0-1 to 0-255.