Sean Sun
12/13/24
CDS DS 210 B1
Professor Chator

Clash of Clans Trophy Prediction Project

This project is centered on the mobile game: Clash of Clans made by Supercell. Released August 2, 2012 Clash of Clans is the 5th highest grossing game on mobile and has generated over 10 billion dollars in revenue. A strategy game focused on bolstering a base and attacking other players' around the world, the main method for measuring success is by how many trophies a player gets which are accumulated by successfully winning attacks against other players' bases and defending your own. I chose a dataset for this game off of Kaggle that consists of 10,001 rows of Clash player statistics from around the world. I decided on this dataset because I have been playing Clash of Clans since around its release in 2nd grade via my Mom's phone and am still playing to this day on my own. I have a long history with this game and have even formed a clan with many of my closest friends which we often talk about and bond over.

Though it may be just a mobile game, it is a complex strategy game that I find much interest in and fighting for trophies is always a constant struggle. The higher trophies a player gets, the better the rewards they reap so I decided on a linear regression model to help predict trophies based on what I thought were the most important factors in predicting it. Those features included "attackWins" which is determined by getting at least one "star" which is either 50% of the base destroyed or destroying the town hall. Secondly "defenseWins" is also determined by this but if the attacker gets zero stars. Next we have donations which is the troop size donations the player *gives* in the clan to other clanmates. I thought that this would be an interesting predictor to analyze because we could study whether or not an active community player who gave troops to others would tend to have higher trophies or not. Finally I selected builder_trophies which is essentially the second base that a player has and whether or not it's doing well.

My code is structured into 3 modules, one called data_reader.rs that takes a file, opens it, and parses through the specified fields, which details I will get into soon. Next is the metrics.rs module that I use to calculate the mean square error, mean absolute error, correlation, and covariance of my regression. Finally I have the main where I convert my features and targets to fit the regression and then compute as well as test at the end.

Going into the code specifics, to start I will delve into the data_reader module. I began by importing my libraries specifically using the fs and io utilities from the standard library. These were essential in helping me read lines efficiently. I then implemented the Debug trait and created a public struct called PlayerRecord that defined my features and target while assigning them the pointer-sized unsigned integer type. I then created the main read_file

function that I would use in the main to open the file and parse through it. I created a new vector to store the parsed vectors in and used the File type from the fs module to open the file and then the BufReader to wrap and read the lines. In a for loop, I first skipped the top row and then unwrapped each line while splitting it and parsing through each indexed field and pushing them to the record vector, returning the final vector at the end.

```rust
// library imports
use std::fs::File; // import the File type from the fs module to help work with files
use std::io::{self, BufRead}; // import IO utilities and BufRead trait for buffered reading

#[derive(Debug)] // implement Debug trait for easy printing
// define a struct to represent a player's data
pub struct PlayerRecord {
    pub attack_wins: usize, // number of attack wins in multiplayer battles
    pub defense_wins: usize, // number of defense wins against multiplayer battle attacks
    pub donations: usize, // number of donations made in a clan
    pub builder_tropies: usize, // number of tropies in the builder hall league
    pub trophies: usize, // number of trophies in multiplayer battles
}

// function to read and parse through dataset from a CSV file
pub fn read_file(path: &str) -> Result<Vec<PlayerRecord>, io::Error> {
    let mut records = Vec::new(); // store parsed player records in a vector

    let file = File::open(path).expect("Could not open file"); // open the file
    let reader = io::BufReader::new(file); // wrap file in buffered reader to read lines efficiently

    for line in reader.lines().skip(1) { // skip the header row
        let line = line?; // unwrap the result of reading a line
        let fields: Vec<&str> = line.split(',').collect(); // split lines by commas into fields

        // parse fields from CSV lineinto PlayerRecord
        let record = PlayerRecord {
            attack_wins: fields[5].parse().unwrap_or(0), //5
            defense_wins: fields[6].parse().unwrap_or(0), //6
            donations: fields[12].parse().unwrap_or(0), //12
            builder_tropies: fields[15].parse().unwrap_or(0), //15
            trophies: fields[10].parse().unwrap_or(0), //10
        };

        records.push(record); // add parsed record to vector
    }

    Ok(records) // return the vector of parsed records if successful

}
```

Next is the metrics module that starts with the ndarray library imports. I decided to calculate the mean absolute error (MAE) and mean squared error (MSE) of my regression because I wanted ways to check the error of my output. I essentially made a separate function for each and went through the mathematical steps on how to calculate MAE and MSE. Iterating through the arrays and summing the absolute differences for MAE and squaring the differences for MSE then dividing by length of the vectors.

```rust
// lib imports
use ndarray::{Array1, Array2};

// calculate Mean Absolute Error (MAE)
pub fn calculate_mae(actual: &Array1<f64>, predicted: &Array1<f64>) -> f64 {
    actual
        .iter()
        .zip(predicted.iter())
        .map(|(a, p)| (a - p).abs())
        .sum::<f64>()
        / actual.len() as f64
}

// calculate Mean Squared Error (MSE)
pub fn calculate_mse(actual: &Array1<f64>, predicted: &Array1<f64>) -> f64 {
    actual
        .iter()
        .zip(predicted.iter())
        .map(|(a, p)| (a - p).powi(2))
        .sum::<f64>()
        / actual.len() as f64
```

I did essentially the same thing for calculating the correlation and covariance of each feature by first creating the public function and then the vector as well as setting up the variables for the mean and standard deviation of the target variable. I iterated through the features using a for loop and computed the correlation and covariance similarly to how I found the MSE and MAE above; I used zip map and sum to iterate simultaneously through arrays, and perform calculations. I then checked if all of the standard deviations were positive because they can never be negative then computed correlation and pushed to the correlations vector.

```rust
/// calculate correlation coefficients between features and target
pub fn calculate_correlations(features: &Array2<f64>, target: &Array1<f64>) -> Vec<f64> {
    let mut correlations = Vec::new();
    let target_mean = target.mean().unwrap_or(0.0);
    let target_std = target.std(0.0);

    for feature_col in features.axis_iter(ndarray::Axis(1)) {
        let feature_mean = feature_col.mean().unwrap_or(0.0);
        let feature_std = feature_col.std(0.0);

        // compute covariance
        let covariance = feature_col.iter()
            .zip(target.iter())
            .map(|(&x, &y)| (x - feature_mean) * (y - target_mean))
            .sum::<f64>() / feature_col.len() as f64;

        // compute correlation
        let correlation = if feature_std > 0.0 && target_std > 0.0 {
            covariance / (feature_std * target_std)
        } else {
            0.0
        };
        correlations.push(correlation);
    }
    correlations
}
```

Below, I have the start of my main module. I start by specifying the uses of my two other modules and the linfa library which will help me with everything I need for the linear regression. I define my file path and match the read_file function in data_reader to use in main. I then check if my data reading errors or not and proceed to create the feature matrix defined as x and the target vector as y. By looping through the records that I created in data_reader, I split up the features into the x matrix and the target trophies into the y vector. Next I shape and prepare the matrix and vector for the regression and print the correlations between the features and the target variable.

```rust
// using my two other modules for reading in the csv and computing MSE and MAE
mod data_reader;
mod metrics;

// libraries
use linfa::Dataset; // data rep for ML
use linfa::traits::Fit; // fitting model trait
use linfa::prelude::*; // general linfa prelude
use linfa_linear::LinearRegression; // implementing linear regression model
use ndarray::Array2; // for manipulating feature matrices

fn main() {
    // file path var
    let file_path = "/Users/seansun/Documents/ds210/project/Sean210Project/body/src/CoC.csv";

    // read data from CSV using data_reader module
    match data_reader::read_file(file_path) {
        Ok(data) => {
            println!("Successfully read data!");

            // Prepare feature matrix (x) and target vector (y)
            let mut x_data = Vec::new();
            let mut y_data = Vec::new();

            // loop through each feature in the record and push to feature matrix and target vector
            for record in data {
                x_data.push(vec![
                    record.attack_wins as f64,
                    record.defense_wins as f64,
                    record.donations as f64,
                    record.builder_tropies as f64,
                ]);
                y_data.push(record.trophies as f64);
            }

            // convert feature and target matrix/vector
            let x = Array2::from_shape_vec((x_data.len(), x_data[0].len()), x_data.concat()).unwrap();
            let y = ndarray::Array1::from_vec(y_data);

            let correlations = metrics::calculate_correlations(&x, &y);
            for (i, correlation) in correlations.iter().enumerate() {
                println!("Correlation between feature {} and trophies: {:.3}", i + 1, correlation);
            }
```

Moving on to the code below, I cloned the matrix and vector data to create a new testing dataset. I then started the linear regression using the dataset and checked if it works correctly. Printing all the results of the regression such as intercepts and predictions. I also added the MAE and MSE predictions from this and printed them to the output. Checking if errors arise in training the model and reading the data at the end.

```
            // create a dataset that's cloned off of the original data
            let dataset = Dataset::new(x.clone(), y.clone());

            // fit linear regression model
            let lin_reg = LinearRegression::new();
            match lin_reg.fit(&dataset) {
                Ok(model) => {
                    println!("Model trained successfully!");

                    // print coefficients and intercept
                    println!("Coefficients: {:?}", model.params());
                    println!("Intercept: {:?}", model.intercept());

                    // predict using the fitted model
                    let predictions = model.predict(&x);
                    println!("Predictions: {:?}", predictions);

                    // calculate MAE and MSE
                    let mae = metrics::calculate_mae(&y, &predictions);
                    let mse = metrics::calculate_mse(&y, &predictions);

                // print metrics
                println!("Mean Absolute Error (MAE): {:.2}", mae);
                println!("Mean Squared Error (MSE): {:.2}", mse);
                    }
                    Err(e) => eprintln!("Failed to train the model: {}", e),
                }
            }
            // if file reading fails, print an error message
            Err(e) => eprintln!("Error reading data: {}", e),
        }
    }
```

In my tests, I have the first two attached in the picture below. The first is to check an expected set of feature and target data. I create their respective arrays and use the correlation, mse, and mae functions to check if they are near their expected values of zero. My second test is checking if a different, simpler file would read successfully, which I performed by creating a temporary data file then using the read_file function and checking if all the file's properties were as to be expected.

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use ndarray::array;
    use tempfile::NamedTempFile;
    use std::io::Write;

    #[test]
    fn test_metrics_calculations() {
        // created a sample dataset
        let features = array![[5.0, 3.0, 10.0, 2.0], [8.0, 6.0, 15.0, 3.0]];
        let trophies = array![200.0, 450.0];

        // test correlation calculation
        let correlations = metrics::calculate_correlations(&features, &trophies);
        assert_eq!(correlations.len(), 4);

        // test MAE calculation
        let predictions = array![200.0, 450.0];
        let mae = metrics::calculate_mae(&trophies, &predictions);
        let mse = metrics::calculate_mse(&trophies, &predictions);

        // assertions checking if the calculated metrics are near zero
        assert!((mae - 0.0).abs() < 1e-6); // both should be near zero
        assert!((mse - 0.0).abs() < 1e-6);
    }

    #[test]
    fn test_read_file_success() {
        // create temporary file that mimics file and stores given 3 lines of data
        let mut temp_file = NamedTempFile::new().unwrap();
        writeln!(temp_file, "header1,header2,header3,header4,header5,attack_wins,defense_wins,dummy1,dummy2,dummy3,trophies,dummy4,donations,(
            .unwrap();
        writeln!(temp_file, "0,0,0,0,0,5,3,0,0,0,1000,0,50,0,0,10").unwrap();
        writeln!(temp_file, "0,0,0,0,0,8,6,0,0,0,1200,0,60,0,0,15").unwrap();

        let result = data_reader::read_file(temp_file.path().to_str().unwrap());
        assert!(result.is_ok()); // check if the file reading is successful, should succeed
        let records = result.unwrap();
        assert_eq!(records.len(), 2); // check if the correct number of records is read
        assert_eq!(records[0].trophies, 1000); // check if the first record's trophies match
    }
```

Moving on to my last test below, we can see that I am essentially doing the opposite of test 2 and checking whether or not the test can catch different types of errors. The first being if the file_path is valid. I initially did this and found that the test worked even if the program panicked at the contents of the data if the file_path was still correct. To combat this and test for it, I looked on ways through stack overflow and saw that if I add a #[should_panic] attribute which will register any panics throughout the file reading process.

```rust
121     #[test]
122     // test file reading failure
123     #[should_panic] // test panic during file reading, aside from invalid path
124     fn test_read_file_invalid_format() {
125         // temporary file
126         let mut temp_file = NamedTempFile::new().unwrap();
127         writeln!(temp_file, "header1,header2,header3,header4,header5,attack_wins,defense_wins,dummy1,dummy2,dummy3,trophies,dummy4,donatio
128             .unwrap();
129         writeln!(temp_file, "invalid,data").unwrap();
130
131         // attempt to read the file
132         let result = data_reader::read_file(temp_file.path().to_str().unwrap());
133         println!("{:?}",result);
134         // check if file reading errors
135         assert!(result.is_err());
136     }
137 }
```

After running these three tests they all pass as can be seen below.

```
seansun@crc-dot1x-nat-10-239-200-128 src % cargo test
   Compiling body v0.1.0 (/Users/seansun/Documents/ds210/project/Sean210Project/body)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.34s
     Running unittests src/main.rs (/Users/seansun/Documents/ds210/project/Sean210Project/body/target/debug/deps/body-d816dd037dc6cc17)

running 3 tests
test tests::test_metrics_calculations ... ok
test tests::test_read_file_success ... ok
test tests::test_read_file_invalid_format - should panic ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Lastly we have the final output. We can see that it first starts out with telling us that the data was read successfully and later that the model was also trained successfully. Moving on to the correlations, we can finally see that each feature's correlation is positive. These can be interpreted as the relationship between each feature and the trophy count with feature 2 (defense wins) being the worst predictor then feature 3(donations) and feature 1(attack wins) in second and third respectively. Out best predictor was builder hall trophies with a correlation of 0.776 meaning someone with higher trophies in the builder base will have expectdly high trophies in the multiplayer base too. These are interesting to look at because we can see that trophies are not heavily reliant on having a good base and winning defenses, but instead better managed through strong attacks and frequent army training through donations. Builder base being the highest predictor is both surprising to me and not surprising. It takes a lot of time to spend training the builder base and is aside from the main base which means time spent gaining trophies in the builder base is not spent on main trophies in the multiplayer base. However, it can be seen that players who play more would naturally have both bases in a strong position.

Moving on to the coefficients, these can be represented as the weights assigned to each feature in my regression model. Essentially, for each unit increase in one feature, trophies is expected to increase by the coefficient. This means that with a coefficient of 51.49, each defense win would predict a 51.49 increase in trophies. This is really interesting because this was found to be the lowest correlation but has the highest coefficient. Meaning while defense is not the best predictor of trophy success, when players do win in their base defense, they gain significant trophies. I find this to be extremely interesting because things like the builder hall trophies are the best predictors, it has a coefficient of just 0.649 meaning each increase in the builder hall trophies gains about 0.649 in the main base. While the best predictors may seem like the best choice, we can see that even bolstering defense yields a significant return despite it being the worst predictor.

Next we have the intercept which essentially means that setting all else equal, the baseline trophies is around 538.93 trophies per player. The predictions are the model's predicted trophy values for players in the dataset. The first player is estimated to have 4320.06, the second to be 5140.42 and so on. While trophies in Clash of Clans are solely integers, this is interesting to see the spread of predictions that the model suggests. Lastly we have the MAE and MSE. The MAE being 467.72 and the MSE being 367,954.90. What a significant difference! The MAE is generally seen as the conditional median of the data and the MSE counterpart is the mean. This suggests that the deviation is extremely large and there may be a problem. I think that in further bolstering of this program, I may try a train_test_split in order to make the difference smaller.

```
     Running  /Users/seansun/Documents/ds210/project/Sean210Project/body/target/debug/body
Successfully read data!
Correlation between feature 1 and trophies: 0.442
Correlation between feature 2 and trophies: 0.318
Correlation between feature 3 and trophies: 0.360
Correlation between feature 4 and trophies: 0.776
Model trained successfully!
Coefficients: [19.641748690840622, 51.49010770463156, 0.23494951827152666, 0.6487660546260452], shape=[4], strides=[1], layout=CFcf (0xf), const ndim=1
Intercept: 538.9267663160404
Predictions: [4320.056623388471, 5140.42111346646, 4173.73961148174, 3743.4136709291383, 3898.29264087725, ..., 2104.0349279021284, 2090.230833356352, 1
940.5231576790845, 1609.7482905531333, 1846.7482915525493], shape=[10001], strides=[1], layout=CFcf (0xf), const ndim=1
Mean Absolute Error (MAE): 467.72
Mean Squared Error (MSE): 367954.90
```

THANKS FOR READING!!!