Computer Science Senior Project

Predicting Ski Racing Results with Machine Learning

Adam Martin

4/27/2017

Table of Contents

Introduction

My name is Adam Martin, and I am a Computer Science and Mathematics double major here at Northern. I came to Northern Michigan University to cross-country ski for the Wildcat ski team. My Freshman year, I took CS120 to satisfy a liberal studies requirement and enjoyed it so much, I decided to add Computer Science as a major. After graduation and professional skiing, I am interested in continuing my education in artificial intelligence and machine learning.

My senior project is to use machine learning to predict world cup ski racing results. Because of my interest in machine learning, I took Andrew Ng's Coursera Machine Learning class during the 2016 summer. I will summarize machine learning as fitting a model to a set of training data in order to predict the outcome for similar data. Therefore, to accomplish my project goal, there are two steps. First, I will collect a lot of data from previous world cup ski races, and second, I will fit a model to this data which explains the outcome of each race. All of the programming is in python 2.

Data Collection Overview

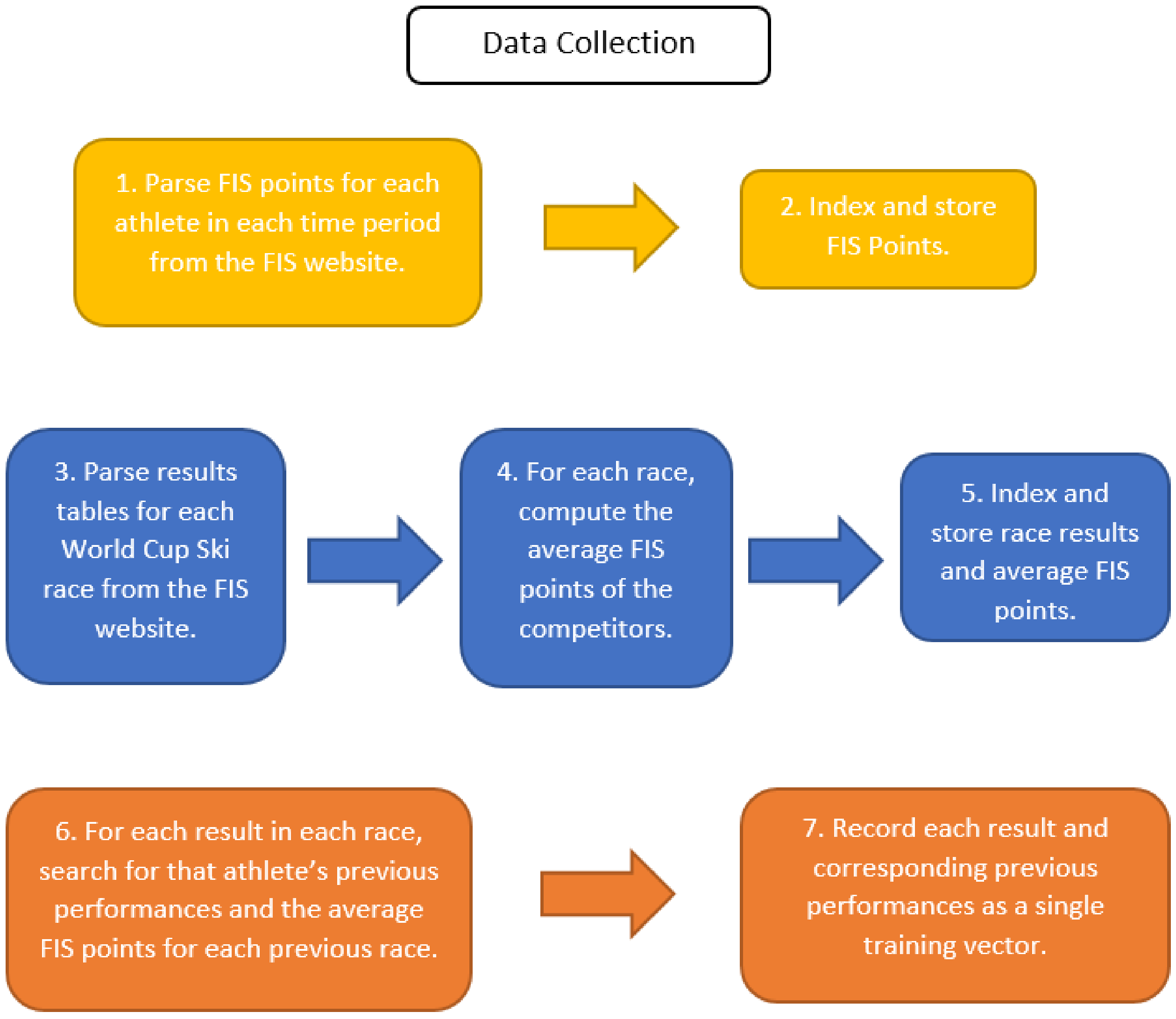
First, let us examine the preliminary process of collecting data. I will explain the details of the machine learning algorithms later, but for now, what is important is that the algorithms will need a series of data points each consisting of multiple independent variables and one dependent variable. Each data point is called a training vector, and the independent variables are called features. If the machine learning is successful, the algorithm will map each set of features to the corresponding value of the dependent variable (or close to it).

To put this into the project context, build backward from our dependent variable. I am trying to predict race results, so the dependent variable is an athlete's outcome in a specific race. The features for this data point can be any number of things, but a simple option is to consider the athlete's outcome in the previous n races. My goal is to predict the outcome of future races, so it is important that every feature is something that could have been determined before the race of our dependent variable took place.

The website www.fis-ski.com hosts pages with race results from all recent skiing world cups. This website provides an obvious source of data for our dependent variable values (race outcomes) and some independent variables (previous performances). At this point, the data is in tables of results for each race, so some significant reformatting is required before it is in the form of a training vector.

Furthermore, to give the algorithm its best possible chances, I hypothesized that the competitiveness of the field in each of the athletes previous races combined with the athlete's previous performances may predict the athlete's outcome better than the athlete's previous performances alone. Fortunately, the FIS website also hosts pages with lists of FIS points, which quantitatively score the racers' ability in a given time period. The FIS points lists need to be combined with the result information before they will be useful.

Now that the objectives are clear, consider a summary of my data collection process. First, I use regular expressions to parse the race result and FIS point data from the html on the FIS website. Then for each race, I compute the average of the competitors FIS points at that time. After this, I store these results tables with the average FIS points of the competitors. Finally, I go through each result from each race and build a training vector, by looking up previous results from the data stored on my computer. The diagram below summarizes this process.


Data Collection
1. Parse FIS points for each athlete in each time period from the FIS website.
2. Index and store FIS Points.
3. Parse results tables for each World Cup Ski race from the FIS website.
4. For each race, compute the average FIS points of the competitors.
5. Index and store race results and average FIS points.
6. For each result in each race, search for that athlete's previous performances and the average FIS points for each previous race.
7. Record each result and corresponding previous performances as a single training vector.

Machine Learning Background Information:

Once we have a set of training data, the next step is to train a machine learning algorithm. In my project, I used three algorithms: linear regression, logistic regression, and a neural network. Before providing an overview of each algorithm, I present some notation to make the explanation of each algorithm more clear.

$$m = \text{the number of training vectors}$$

$$n = \text{the number of features in each training vector}$$

$$x_j^{(i)} = \text{the } j^{th} \text{ feature of the } i^{th} \text{ training vector}$$

$$y^{(i)} = \text{the outcome of the } i^{th} \text{ training vector}$$

Linear Regression

In linear regression, we hypothesis that the outcome of each training vector is approximately a linear combination of each feature.

$$a_0 + a_1 \cdot x_1^{(i)} + \cdots + a_n \cdot x_n^{(i)} = h_A(x^{(i)}) \approx y^{(i)}$$

Therefore, the goal is to find a set of coefficients $a_0, a_1, \ldots, a_n$ that do the "best" job approximating each $y^{(i)}$. To outline this more formally, define a cost function, which maps a set of coefficients to a number describing how well the coefficients fit the training data.

$$J(A) = \frac{1}{2m} \cdot \sum_{i=1}^{m} \left(h_A\left(x^{(i)}\right) - y^{(i)}\right)^2$$

The cost function is ½ the average square error of each training vector with the current set of coefficients. With this machinery defined, our goal is to find the coefficients A that minimize the cost function.

Gradient descent is one method used to find the minimum of the cost function. Gradient descent begins with a random set of coefficients and adjusts each coefficient over many iterations to find a local minimum. Specifically, on every iteration, each coefficient is adjusted by a multiple of its partial derivative.

$$a_j = a_j - \alpha \frac{1}{m} \cdot \sum_{i=1}^{m} (h_A(x^{(i)} - y^{(i)}) \cdot x_j^{(i)}$$

By examining the cost function during this process, it is evident when the cost function is close to its minimum because it will only change a small amount on each iteration.

Once we determine the optimal coefficients A, they can be used to predict the outcome for any feature vector. Note that linear regression deals with a quantitative response variable.

Logistic Regression

Logistic regression differs from linear regression by predicting boolean response variables. The hypothesis in logistic regression is similar to linear regression with an extra step added.

$$z^{(i)} = a_0 + a_1 \cdot x_1^{(i)} + \cdots + a_n \cdot x_n^{(i)}$$

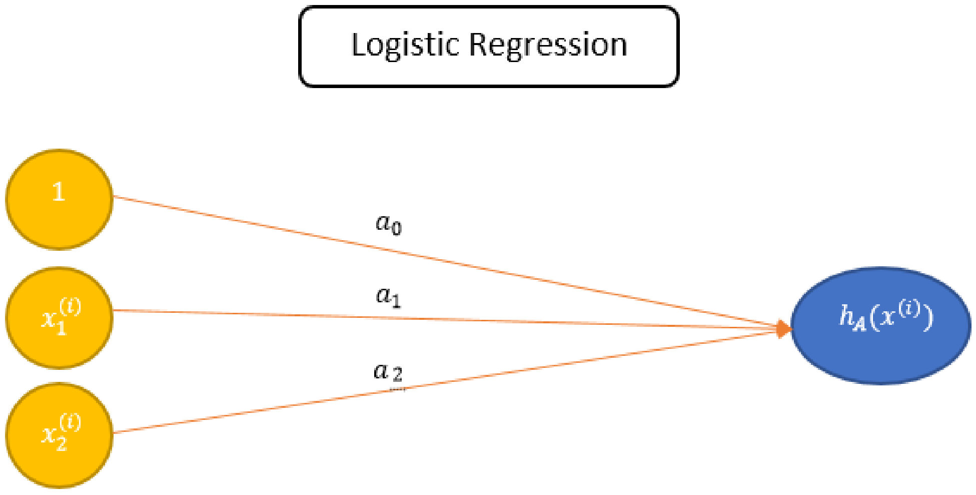
$$y^{(i)} \approx h_A\left(x^{(i)}\right) = \frac{1}{1 + e^{-z^{(i)}}}$$

Consider that if z is a large positive number, $h_A(x^{(i)})$ will be approximately 1 and if z is a large negative number $h_A(x^{(i)})$ will be approximately 0.

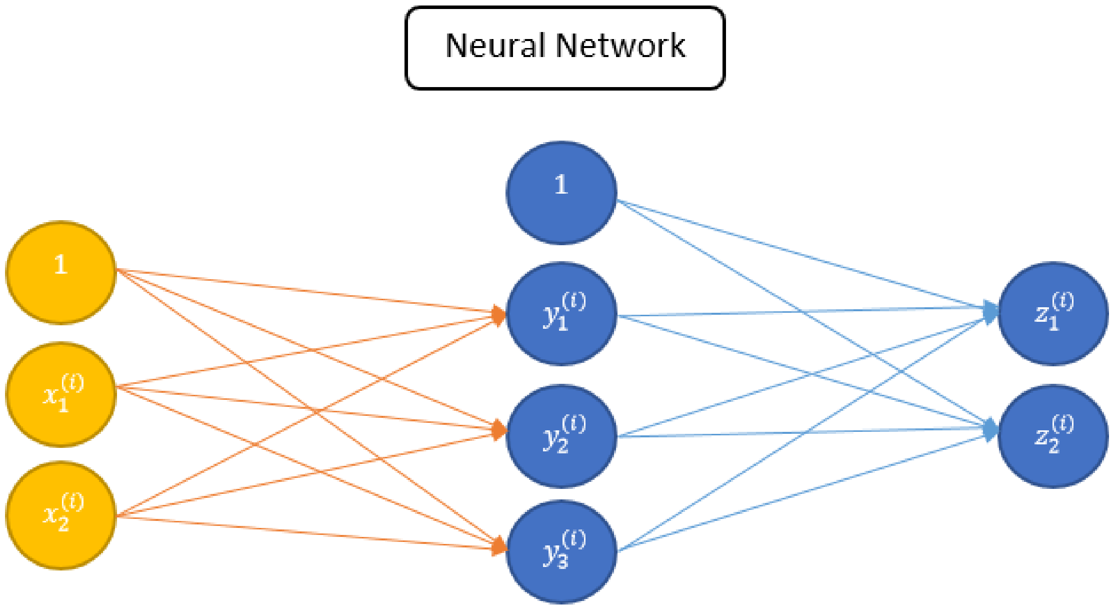
A logistic regression algorithm is trained similarly to a linear regression algorithm: by defining a cost function and using gradient descent to minimize this function.

Neural Networks

Before looking at neural networks, consider a visual representation of logistic regression with two features.



This logistic regression model is actually a two layer neural network. Neural networks are stacked layers of logistic regression models with more than one output node. Below is a three layer neural network with two input nodes, three middle nodes, and two output nodes.



In the diagram, $y_1^{(i)}$, $y_2^{(i)}$, and $y_3^{(i)}$ can each be thought of as a logistic regression of the features $x_1^{(i)}$ and $x_2^{(i)}$. Similarly, $z_1^{(i)}$ and $z_2^{(i)}$ are each a logistic regression of the artificial features $y_1^{(i)}$, $y_2^{(i)}$, and $y_3^{(i)}$. The partial derivatives of the coefficients for neural networks are defined by working backward from the output nodes (at the right) to the input features (at the left).

Use of Regular Expressions

Now that we have covered an outline of my process for collecting data and the algorithms I will use to analyze the data, I will present a few interesting and noteworthy portions of my project in more detail.

First, in order to collect the race result and FIS point data, it was necessary to automate the process. I accomplished this with regular expressions. My strategy was to use each regular expression for a very specific task and then use python and more regular expressions to break down the results further.

A common pattern in my regular expression usage was to extract the text between two html tags. Below is a snippet of my code.

```python
def getElementContent(element, html):
    # Expects an html element name, and some html to parse
    # Returns a list of the content between each pair of html tags
    # Note: This will not recognize nested tags of the same kind.

    reo = re.compile(reNotAComment + '<' + element + reRestOfTag + \
        reCapture(reAllNotGreedy) + reGroup('</' + element + '>'))
    return reo.findall(html)
```

As shown in the example above, I store some common regular expression patterns as variables, which makes long regular expressions easier to understand. The function of the regular expression above is to look for an html tag and then capture everything after it until the matching end tag. For reference, here is the regular expression with 'table' passed in as an argument and other variable values substituted for variable names.

```
(?<!<!--)<table[^>]*>([\d\D]*?)(?:</table>)
```

Data Storage

Once I parse the html from the FIS web pages into tables of FIS points and race results, I store the data so that I can extract training vectors from it in the future. Initially, this data is in the form of two dimensional python lists. The usage of the data will be storing it a single time and repeatedly looking up how a single athlete performed in a race or what a single athlete's points are. Considering this, it makes sense to store each race result and FIS points list as a python dictionary since access performance for dictionaries is on average constant time in python. (Python dictionaries are implemented with a hashmap.) Once the data are in dictionaries, I store them in .json files.

Another important consideration for data storage is indexing. As I store hundreds of race results and points lists, I also update indexing lists, which keep track of some information about each file including its file name. This means that later, when I need to access one of these files, I can search for the file in the index and precisely open the correct file.

Building Training Vectors

After the dictionaries with race result data are stored in files, everything is in place to build training vectors for the machine learning algorithms. As discussed before, the goal is to build a vector with one dependent variable (an athlete's outcome in a specific race) and multiple features that may explain the dependent variable (the same athlete's performance in previous races).

Beyond this generality, it is desirable to provide functionality to build different types of training vectors. For example, we could limit the type of races we are trying to predict like considering men and women separately or only considering interval start, distance races. Additionally, we should be able to select exactly what dependent variable we want to consider. Is a racer's final place or percentage behind the winner's time under consideration? Furthermore, there are an endless combination of features, which can be used in the feature vector.

Fortunately, functions are first class objects in python. My solution was to build a generic algorithm that builds training vectors and have this algorithm accept other functions as arguments. For example, rather than have the algorithm always select an athlete's rank as the dependent variable, the algorithm defers this to a parameter function. The parameter function expects a result object and returns the value of the dependent variable. Depending on the function passed into the algorithm, it can return the athlete's rank, percentage back, or a 0 or 1 indicating if they placed in the top 5.

The function below is an example of a factory that produces functions which return whether a result was in the top ‘x’ places.

```
def getFuncIsTop(x):
    # Returns a function which will determine if a result is in the top x places

    def isTop(result):
        # Returns 0 or 1 whethor result is in the top x places

        return int((not result is None) and getRank(result) <= x)

    return isTop
```

Matrix Operations to Accomplish Machine Learning

Now that we have training data, we can begin with the machine learning portion of the project. I used the numpy python library to implement the machine learning algorithms. Numpy contains a tool set to define and manipulate matrices. This is important because simplifying the algorithms to matrix operations means that the optimized numpy routines will perform the computation intensive work.

Continuing with the notation from the machine learning background information section, I apply the notation to the matrices.

$$\text{Matrix of training data: } X = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \vdots & & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}$$

$$\text{Column vector of training vector outcomes: } \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

$$\text{Column vector of coefficients (theta): } \theta = \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix}$$

We define a cost function, which maps these arguments to the linear regression cost and a column vector with the partial derivatives of the cost function with respect to each coefficient.

```
def cost(X, y, theta, reg):
    # Expects training matrix (X), column vector of labels (y),
    #   column vector of coefficents (theta), and regularization constant (reg)
    # Returns Cost(theta) = 1/(2m) * SUM((predictedValue - actualValue)^2)

    m = len(y)
    h = np.dot(X, theta)
    d = h - y
    J = np.sum(d**2) / (2*m) + np.sum(theta[1:]**2) * reg / (2*m)
    grad = np.dot(np.transpose(X), d) / m
    grad[1:] += theta[1:] * reg / m
    return J, grad
```

In the function above, 'np.dot' performs matrix multiplication and 'np.sum' finds the sum of all the values in a matrix. The interested reader can apply the definitions of X, y, and theta on page 7 to the function and verify that J is equal to the correct cost. In this process, the reader should ignore the terms involving the regularization constant labeled 'reg' as regularization is beyond the scope of this summary.

Once the partial derivatives of the cost function are derived, implementing gradient descent is a trivial programming exercise.

Feature Expansion

One of the coolest functions I wrote in my project is for feature expansion. The linear regression model defined earlier was limited to a linear combination of each of the features. However, linear regression becomes much more powerful when multiples and powers of the original features are added to the model. For example we can create a complete second order model from two features, x_1 and x_2, by using the features $1, x_1, x_2, x_1^2, x_1x_2$, and x_2^2. The second order model has the capacity to fit non-linear trends in the data.

My objective was to create a function that would map a training matrix and maximum degree to a new training matrix including the nonlinear features. If the maximum degree is known ahead of time, this is not an especially difficult task, but if the maximum degree is provided as a parameter, it appears that the program needs a variable number of for loops. I accomplished this with recursion by having each activation call the recursive function inside of a for loop.

The parameter 'powers' is a list built to contain the maximum degree of the features after and including each index. Going back to the example with original features $[x_1, x_2]$, a list [2, 1] would represent the expanded feature x_1x_2.

```
def expandFeatures(X, degree):
    # Expects training matrix (X)
    #   and the maximum degree of each feature in the expanded training matrix (degree)
    # Returns a complete degree order training matrix

    def recExpandFeatures(X, powers, degree):
        # Recursive instrument
        # Expects a list of the total degree of the following features (powers)

        if len(powers) == X.shape[1]:
            col = np.ones(X.shape[0])
            for i in range(X.shape[1]-1, -1, -1):
                power = powers[i] - np.sum(powers[i+1:])
                powers[i] = power
                col = col * (X[:,i] ** power)
            return np.transpose([col])

        output = np.empty((X.shape[0],0))
        for i in range(0, degree+1):
            partial = recExpandFeatures(X, powers + [i], i)
            output = np.concatenate((output, partial), axis=1)
        return output

    return recExpandFeatures(X, [], degree)
```

Outlier Detection

I also built the capacity for outlier detection to improve the quality of training data. To explain what an outlier is, among the human race, Batman's degree of manliness is an outlier. In the context of my project, there are two situations where outliers have the potential to hinder the model training.

First, when building training vectors, it is possible one of the athlete's previous races was uncharacteristically bad (maybe they fell or broke a piece of equipment). In this situation, that race will probably not be especially helpful in predicting a future performance. I try to resolve this issue, by examining the variation in training features and substituting older races for the uncharacteristic features.

Secondly, the dependent variable, race outcome, itself can be an outlier. (For similar reasons any of the features may have been outliers.) I address this situation, by training the machine learning algorithm twice. After an initial run, I determine a small number of training vectors with uncommonly high error. I assume these training vectors are outliers and retrain the model on the original data minus the set of outliers.

Results

In my project proposal, I stated two prediction goals. The first challenge was to predict a racer's percent back in an individual start race. After training a linear regression model, I achieved an average error margin of 1.9 percent back. In a 25-minute race, this will correspond to predicting a racers time behind the winner within 28 seconds. For perspective, most racers will finish within 5 to 6 minutes of the winner.

The second objective is to predict a racer's place. A neural network implemented to predict an athlete's place within categories of top 3, top 8, top 15, top 30 and higher than 30, classified the examples in the correct group 54 percent of the time and classified examples within 1 group of the correct group 83 percent of the time.

Overall, I think these are satisfactory results. Ski racing is an unpredictable sport; variables such as course profiles, snow conditions, and varying athlete fitness make forecasting results difficult, so the results above represent non-trivial predictions.

Conclusions

I really enjoyed this project and believe I accomplished all of the initial objectives. I learned about machine learning, devised a scheme to collect data on World Cup Skiing, and trained three machine learning algorithms to predict the data. The biggest unanticipated challenge for me was organizing a system to collect, store, and index the race result and FIS points data. For future improvement, I think including features, which represent the racecourse and terrain of each competition, will help explain more of the variability in results. Overall, I learned a lot by building this project gained machine learning experience.

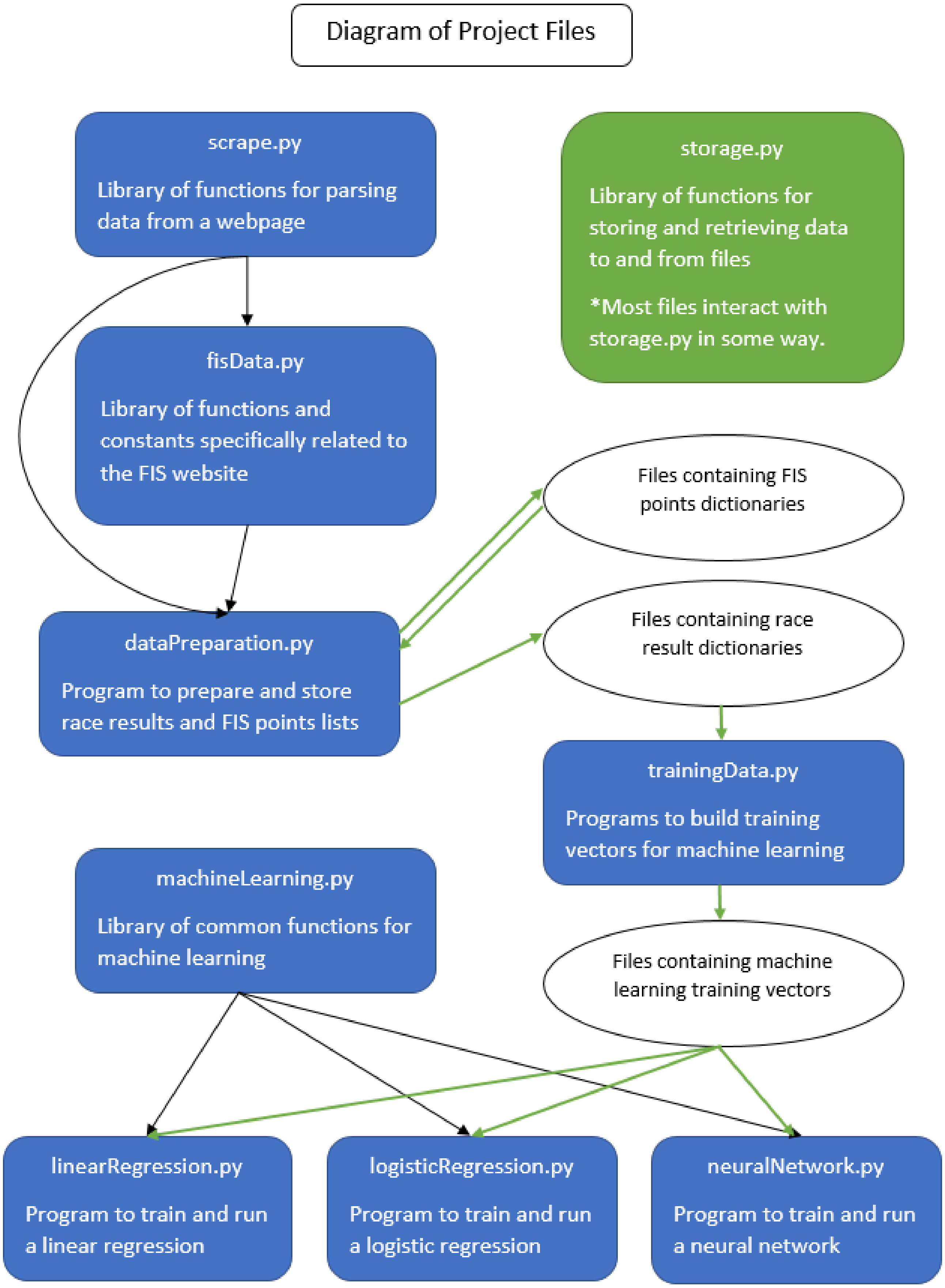


Diagram of Project Files
scrape.py
Library of functions for parsing data from a webpage
storage.py
Library of functions for storing and retrieving data to and from files
*Most files interact with storage.py in some way.
fisData.py
Library of functions and constants specifically related to the FIS website
Files containing FIS points dictionaries
Files containing race result dictionaries
dataPreparation.py
Program to prepare and store race results and FIS points lists
trainingData.py
Programs to build training vectors for machine learning
machineLearning.py
Library of common functions for machine learning
Files containing machine learning training vectors
linearRegression.py
Program to train and run a linear regression
logisticRegression.py
Program to train and run a logistic regression
neuralNetwork.py
Program to train and run a neural network