Machine Learning Capstone Project Report

Title: Predicting Formula One Outcomes

Overview:

In this paper I will use Machine Learning algorithms to try and predict the outcome of Formula One races. As one of the top motor sports in the world, Formula One teams are often very heavily invested, and usually provide millions of pounds towards research and development of their cars, engines, brakes etc. Teams and drivers are often result driven and employ a number of techniques to try and get an edge up on the competition. This project was inspired by a personal interest into F1 racing. In this project, outcome refers to the winner of the race, rather than predicting position by position how the drivers will finish.

The dataset I will be using is the Formula One race data available here (with a Kaggle account): https://www.kaggle.com/cjgdev/formula-1-race-data-19502017. This dataset contains information from 1950 to 2018, including races, results, lap times, locations etc. It does not however contain any data on weather conditions at the time of the race, nor does it contain information on tires used throughout the race. Unfortunately, this will create some inaccuracies in our data model; however it cannot be avoided at the present moment. (Attempts to source weather information proved futile, and it is likely I will never know what the weather was like in Oporto, Portugal on the 24th of August 1958 !!)

Problem Statement:

Predict, as accurately as possible, the winner of a Formula One race based on factors such as starting grid position, location, qualifying times etc. This problem is typical of a supervised learning problem and as such I will develop supervised learning models using classification algorithms to fit the data. Classification models are chosen here rather than regression as the data is discrete.

Data Analysis:

The data is divided into several different CSV files, each CSV having a unique ID column to correctly identify records across data sets e.g. the driverId column in the results.csv corresponds to the driverId column in the drivers.csv file. The individual files first needed to be merged to provide a comprehensive data set.

qualifying.csv:

qualifyId: Unique ID of the qualification data

raceId: Unique ID of the race

driverId: Unique ID of the driver
constructorId: Unique ID of the constructor
position: Grid position of the driver after qualifying
q1: Qualifying one time in minutes
q2: Qualifying two time in minutes
q3: Qualifying three time in minutes

races.csv:
raceId: Unique ID of each race
circuitId: Unique ID of the circuit

results.csv:
resultId: Unique ID of the result
raceId: Unique ID of the race
driverID: Unique ID of the driver
constructorId: Unique ID of the constructor
grid: Starting grid position of the driver
position: Finishing position of the driver
points: Points earned by the driver
laps: Laps completed by the driver
milliseconds: Time taken for the race in milliseconds
fastestLap: Fastest lap
fastestLapTime: Time of the fastest lap
fastestLapSpeed: Fastest speed
statusId: Unique ID of status

The three data sets that we are interested in are results.csv, which contains the result of each race, qualifying.csv which contains the data on the qualification for each race and races.csv which contains data on each race e.g. date and time, circuitId. The data first needed to be merged into one dataset; I did this by first joining the results to the qualifying data along common columns (raceId, driverId, and constructorId) and then joining this to the race data along the raceId column. All of these operations were simply done using various functions offered by pandas. (read_csv, rename, merge etc). Finally, I had a dataset with 23,777 rows and 19 columns to work with. A sample of the dataset is shown below.

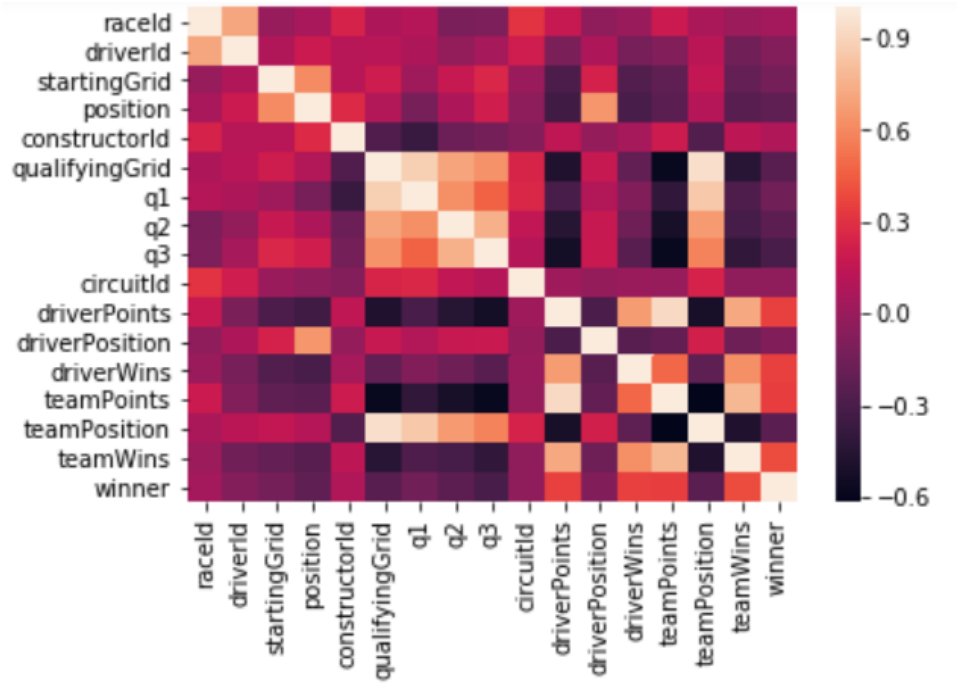
	raceId	driverId	startingGrid	position	constructorId	qualifyingGrid	q1	q2	q3	circuitId	driverPoints	driverPosition	driverWins
0	1	18	1	1.0	23.0	1.0	1:25.211	1:24.855	1:26.202	1	10.0	1.0	1.0
1	1	22	2	2.0	23.0	2.0	1:25.006	1:24.783	1:26.505	1	8.0	2.0	0.0
2	1	15	20	3.0	7.0	8.0	1:26.194	1:25.265	1:27.127	1	6.0	3.0	0.0
3	1	10	19	4.0	7.0	6.0	1:25.499	1:25.281	1:26.975	1	5.0	4.0	0.0
4	1	4	10	5.0	4.0	12.0	1:26.026	1:25.605	NaN	1	4.0	5.0	0.0

Unfortunately, as with any dataset, some of the data was incomplete or missing (e.g we can see a NaN entry under the q3 column above), so I first needed to perform some initial pre-processing and cleaning to tidy it up. The first addition was the creation a new column called winner which would simply track whether the driver won that particular race, and was derived from the position column. If the position was 1, then the winner column was set to true otherwise false. Secondly, due to some missing data, it was necessary to remove some records. In this case, I felt that it was appropriate that if any of the records were missing for qualifyingGrid, constructorId, q1, q2 or q3 where the driver had won, then those records should be removed to avoid giving the model inaccurate data. The number of these records was only a small sample and was negligible to the final dataset. Once these were removed, I could backfill the remaining entries with data associated only to a loss. For example, since no constructor has an ID of zero, we associate any null constructorId to zero. This is not troublesome as we removed any records that a winner had no constructorId. Similarly, for an empty qualifyingGrid, we assign it the max qualifying grid position from the data. This is done using the fillna method for a dataframe. Lastly, the columns q1, q2, q3 needed to be cast from strings into milliseconds containing the qualifying times. A snippet of the cleaned and ready data can be seen below, along with the results of dataframe.describe which gives us some additional meta data.

	raceId	driverId	startingGrid	position	constructorId	qualifyingGrid	q1	q2	q3	circuitId	driverPoints	driverPosition	driverWins
0	1	18	1	1.0	23.0	1.0	85211	84855	86202	1	10.0	1.0	1.0
1	1	22	2	2.0	23.0	2.0	85006	84783	86505	1	8.0	2.0	0.0
2	1	15	20	3.0	7.0	8.0	86194	85265	87127	1	6.0	3.0	0.0
3	1	10	19	4.0	7.0	6.0	85499	85281	86975	1	5.0	4.0	0.0
4	1	4	10	5.0	4.0	12.0	86026	85605	129776	1	4.0	5.0	0.0

	raceId	driverId	startingGrid	position	constructorId	qualifyingGrid	q1	q2	q3	circuitId
count	23021.000000	23021.000000	23021.000000	12471.000000	23021.000000	23021.000000	2.302100e+04	23021.000000	23021.000000	23021.000000
mean	488.106251	227.316711	11.542722	8.193409	11.304591	22.737631	1.336330e+05	125690.941705	125956.369489	21.682203
std	271.180739	233.037665	7.289625	4.574199	38.835073	8.458940	3.117001e+04	16321.810748	12372.373062	15.812656
min	1.000000	1.000000	0.000000	1.000000	0.000000	1.000000	6.506400e+04	64316.000000	64251.000000	1.000000
25%	274.000000	55.000000	5.000000	4.000000	0.000000	18.000000	9.870300e+04	132470.000000	129776.000000	9.000000
50%	478.000000	154.000000	11.000000	8.000000	0.000000	28.000000	1.538850e+05	132470.000000	129776.000000	18.000000
75%	720.000000	314.000000	17.000000	11.000000	4.000000	28.000000	1.538850e+05	132470.000000	129776.000000	31.000000
max	988.000000	843.000000	34.000000	33.000000	210.000000	28.000000	1.002640e+06	132470.000000	129776.000000	73.000000

It is also often quite interesting to produce a correlation matrix of the data to get a flavour of how the data in one column is relevant to data in another column. We can do this quite easily using dataframe.corr and seaborn.heatmap to get an output, shown below:



The correlation graph above has some surprising features. We can see a highly correlated square in the centre of the grid corresponding to qualifyingGrid, q1, q2 and q3. This is not surprising since a faster qualifying time means a higher grid position. These four attributes are also highly correlated to teamPosititon, which again isn't too surprising since if a team is performing well, it is likely they have better driver and cars which would lead to better times on track.

However, it is interesting to note that these attributes are not strongly correlated to the feature we are interested in, winner. From the graph, it appears that winner is strongly correlated to teamWins, teamPoints, driverWins and driverPoints. These attributes are in themselves not surprising, however worth noting that other seemingly important factors (e.g. driverID, startingGrid and circuitId) are not strongly correlated.

This would lead us to believe that a driver and a team performing well stand more of a chance of winning any given race, rather than a particular driver starting from the front of the grid on his favourite circuit.

There are some other pockets of correlation dotted on the matrix, e.g. driverWins with teamWins and teamPoints, position and startingGrid. These are not surprising results, but it is still good to see that the data backs up our intuition.

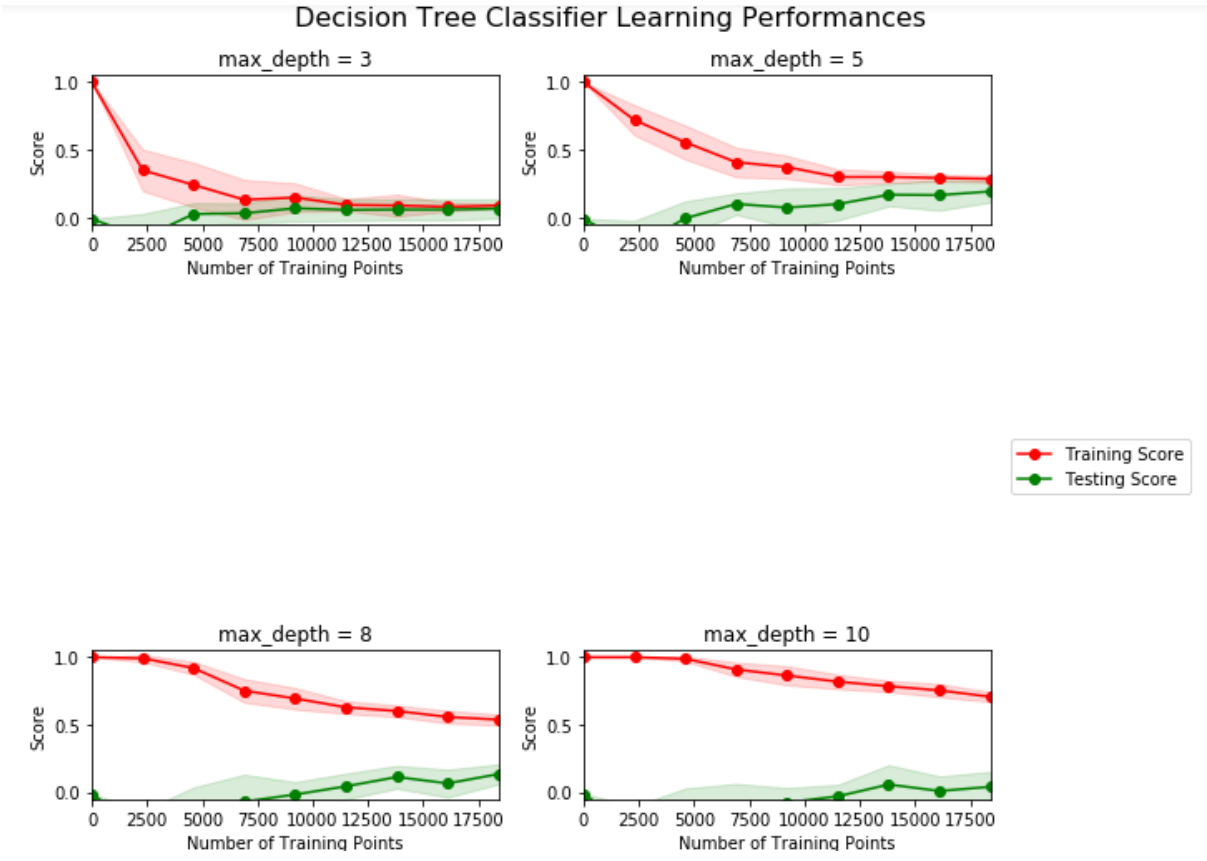
Benchmark Models:

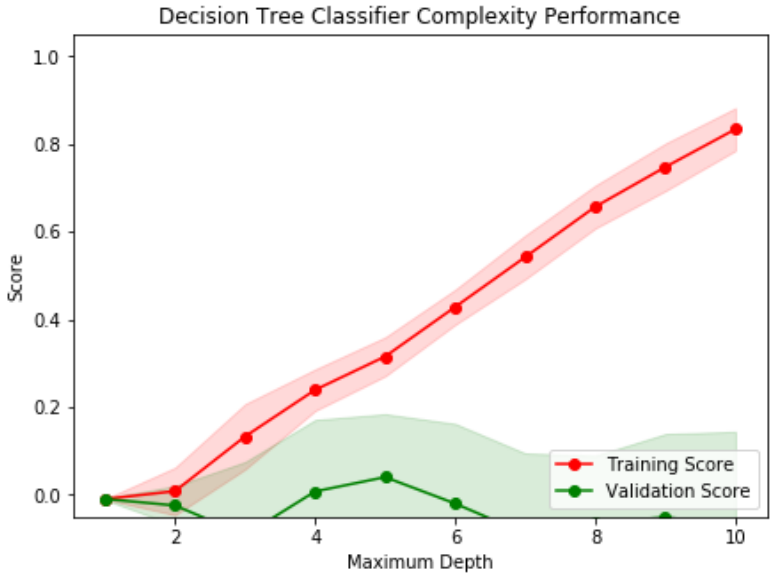
As an initial benchmark model to get a feel of how well we should expect our model to do, consider that each F1 race has approximately 20 participants. (As of 2015, there are 20 drivers per race; however previous to this, races often had more. 20 will serve as a good approximation for us here.) Since each race can only have one winner, this gives a 1 in 20 or 5% chance of any driver winning. Conversely, it gives a 19 in 20 or 95% chance of not winning. (Of course, I am assuming that the winner is randomly distributed here, which is not

the case.) So, in this case if were to naively predict that a driver was to lose, we would be correct 95% of the time. This is quite a high accuracy rate, but we are hoping to find a model that regularly guesses correctly more than 95% of the time. Doing the actual calculation on the data gives an accuracy score of 0.9903 and an F score of 1. This is a very accurate model, but provides us with no real insight. We will need to use a more sophisticated model. Here goes.

More Advanced Models:

The first model I will try is the simple Decision Tree Classifier. I first split the data into the result of the race (winner column from the data) and the features that are likely to have contributed to the result (everything else except winner and position columns). Next using train_test_split from sklearn, we divide the dataset into training and testing subsets. I choose an 80/20 percent split giving me a training set consisting of 18416 records and a testing set of 4605 records. At this point, it is worth plotting a few graphs to represent the training and testing scores for the Decision Tree Classifier at different max depths. This is done using ModelLearning and ModelComplexity from visuals.py.





We can see from the top graph that as the max_depth parameter increases, so too does the training score. This indicates that the optimal max_depth parameter is more than 10, and in order to find it we use GridSearchCV. Through trial and error, I discovered that the optimal value was 374. So we use that in our model and calculate the r2 score. The r2 score is the coefficient of determination, and in this case it is 0.8143. The accuracy for this model is 0.9982 and the F-score is 0.9051. Versus the naïve model, we have traded increased accuracy for a decreased F-score. Since the F-score is still very high I think is a worthwhile trade off, given the naivety of the naïve model.

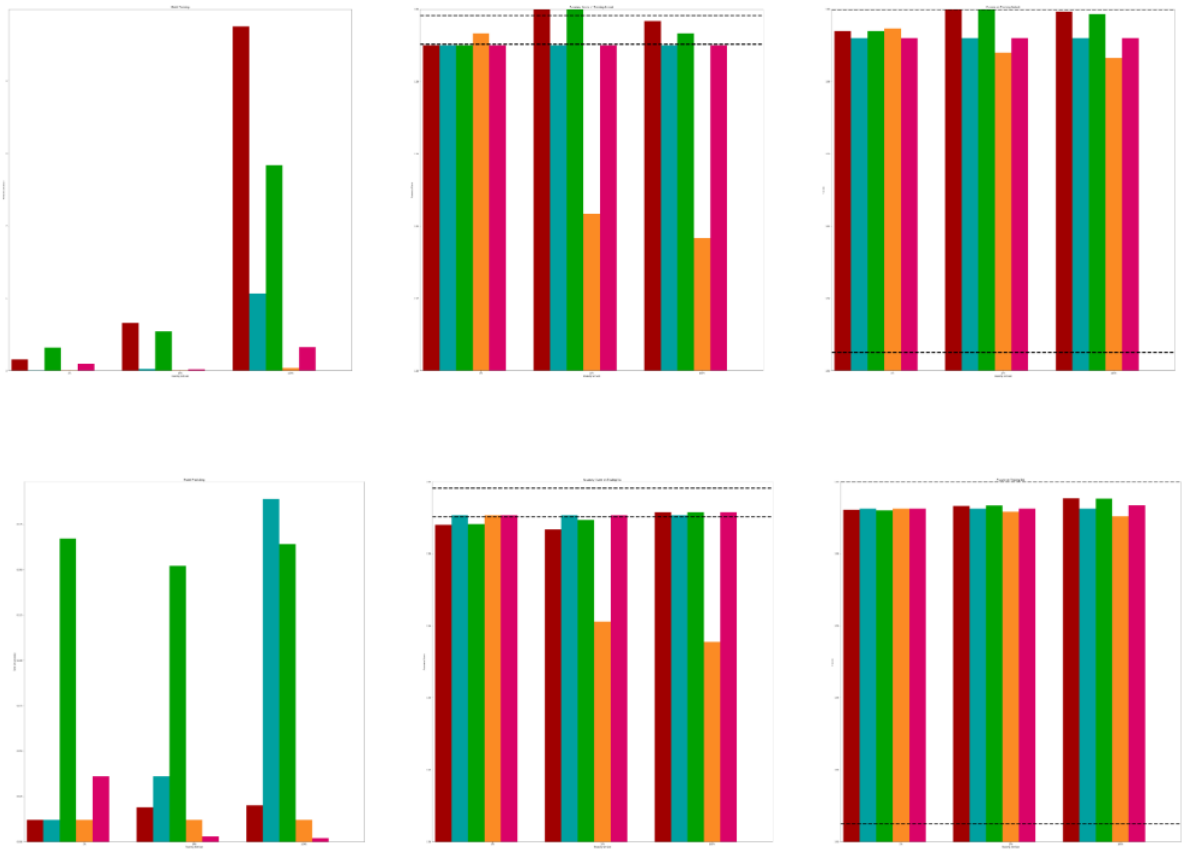
Perhaps we can make even more improvements by using another model from the sklearn family. Before doing this, it is worth remembering that it is considered good practice to normalize our numerical features. Applying a scaling to the data does not change the shape of each feature's distribution; however normalization ensures that each feature is treated equally when applying supervised learners. Note that once scaling is applied, observing the data in its raw form will no longer have the same original meaning, as exampled below. We use sklearn.preprocessing.MinMaxScaler for this.

	raceId	driverId	startingGrid	constructorId	qualifyingGrid	q1	q2	q3	circuitId	driverPoints	driverPosition	driverWins	teamPoints
0	0.0	0.020190	0.029412	0.109524	0.000000	0.021488	0.301362	0.335002	0.0	0.025189	0.000000	0.076923	0.023529
1	0.0	0.024941	0.058824	0.109524	0.037037	0.021270	0.300305	0.339626	0.0	0.020151	0.009615	0.000000	0.023529
2	0.0	0.016627	0.588235	0.033333	0.259259	0.022537	0.307377	0.349119	0.0	0.015113	0.019231	0.000000	0.014379
3	0.0	0.010689	0.558824	0.033333	0.185185	0.021796	0.307612	0.346799	0.0	0.012594	0.028846	0.000000	0.014379
4	0.0	0.003563	0.294118	0.019048	0.407407	0.022358	0.312366	1.000000	0.0	0.010076	0.038462	0.000000	0.005229

To properly evaluate the performance of the model we choose, it is important to create a training and predicting pipeline that allows us to quickly and effectively train models using various sizes of training data and perform predictions on the testing data. I created a function called train_predict to do this. It trains and tests a model on various different sample sizes and stores the results. We can call this function iteratively on many different models to evaluate their usefulness. I ran the following models through the pipeline and plotted their results for

time taken, accuracy score and F-score on 1%, 10% and 100% of the sample training data. The top three graphs are the training results and the bottom three are the testing results.

- GradientBoostingClassifier (Red)
- SVC (Blue)
- AdaBoostClassifier (Green)
- GaussianNB (Yellow)
- LogisticRegression (Pink)



As we can see from the graphs above all models are giving very high accuracies. All that remains is to choose a model and then use GridSearchCV to fine-tune the parameters. I choose to use the AdaBoostClassifier since it has a very high accuracy level and has also been used to analyse race data before at http://www-math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Eric-Boosting304FinalRpdf.pdf. Adaboost has the advantage that is a fast algorithm, by mainly focusing on the incorrectly classified points on each iteration and assigning them a higher weight. The disadvantages of the AdaBoostClassifier are that it is sensitive to complex data and outliers in the data. This is also a good model to use on our dataset since we have a large dataset but it is not too complex. We will be able to carry out multiple training iterations on the data to achieve more accuracy overall.

Running the AdaBoostClassifier through GridSearchCV and using trial and error to determine the optimal list of parameters to test we get the following:

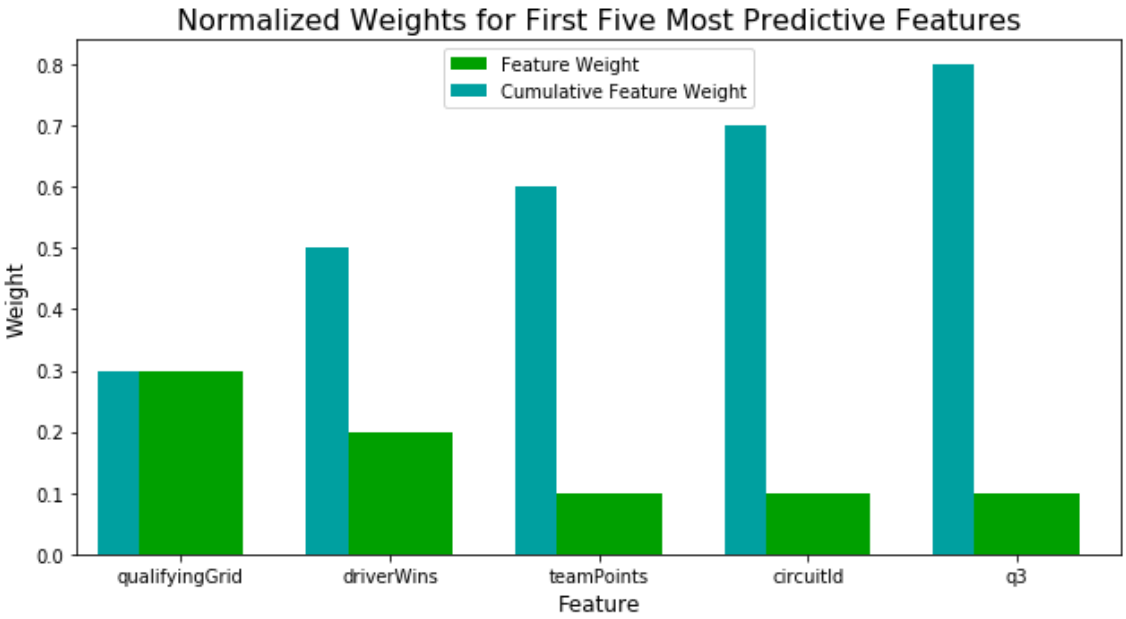
	Accuracy Score	**F-Score**
Un-optimised Model	0.9915	0.5348
Optimised Model	0.9924	0.5967

Recall the accuracy and F-scores for the naïve and Decision Tree models:

	Accuracy Score	**F-Score**
Decision Tree Classifier	0.9982	0.9051
Naïve Model	0.9903	1.0000

So we see that although the AdaBoostClassifier trumps the naïve model on accuracy, it still falls short of the Decision Tree. The F score also lies below that of both previous models. The AdaBoostClassifier is still a very accurate model in its right, accurate over 99% of the time!

Finally, even though the AdaBoostClassifier didn't best the Decision Tree, it is still an interesting exercise to see which features contributed most to the predictions. We can do this by using the feature_importances_ of a model, and we plot them along with the cumulative feature weight.



The graph above tells us that the top five most important features were qualifyingGrid, driverWins, teamPoints, circuitId and q3. When compared to the correlation matrix above, this yields some surprising results. For example, qualifingGrid, circuitId and q3 were not strongly correlated to the outcome of the race but yet, qualifying Grid is the most important feature of the model. (In a sense, this is not surprised since the leader at the beginning of the race has the biggest adavantage from the start.) We also see driverWins and teamPoints

appear as the 2nd and 3rd most important feature and this agrees with the correlation matrix. It is also interesting to note that q3 appears rather than q2 or q1. This is because all drivers compete in q1 and register a time. Then the slowest 5 drivers are dropped and the 15 remaining register a new time. Again, the remaining 10 register a time for q3 and this determines the qualifying grid position. Hence, it makes sense that drivers with a lower q3 time (compared to no q3 time) are likely to be better drivers and have a better chance of winning the race outright. It is alos interesting that qualifying grid appears rather than starting grid. (The driver may have incurred a penalty for some incident, however the model takes care of this.)

The graph above indicates that a driver with a high qualifying grid position, with a lot of wins under his belt, driving for a team doing well and on a circuit he likes, is most likely to win a given race.

It is also interesting to see if we can create a model using only these five features, which still gives an accurate result but saves us some computation time by ignoring possibly negligible features. We use clone for this purpose. Creating the cloned model and training on the five most important features:

	Accuracy Score	**F-Score**
Full Data model	0.9924	0.5967
Reduced Data Model	0.9924	0.5897

So it seems that we can create a slimmed down model using only these five features which is as accurate as a model trained on the whole dataset. However, the f-score decreases by seven parts in a thousand which is a small price to pay for a computationally less intense model.

Conclusion:

At this stage I have produced two highly accurate models for predicting the outcome of a Formula One race, which given some new input data would produce a prediction of the outcome which would be accurate over 99% of the time. On reflection, I believe that the naïve model only had such a high accuracy rate because I was forced to remove some of the winning data at the beginning. This obviously decreased the percentage of winning records in the dataset and drove up the accuracy of predicting a non-win. I would have expected the actual percentage to be closer to 95%. It is likely that this also affected the ‘learning’ of both the Decision Tree and AdaBoostClassifier models too.

It is worth noting that both models are very simplistic in that they only produce a yes or no prediction for each record. In real life scenarios, we would expect that if the model was given information relating to the 20 participants in the next F1 Grand Prix, the model would predict one of the 20 to win, however it currently does not do that. I talk about this and a few other possible improvements in the next section.

Lastly, while this is not exactly ground breaking research, as an initial soiree into the world of Machine Learning, I am quite happy with the models and results I have produced in this paper, and will certainly be applying them to the upcoming Formula One race season.

Further Improvements:

- It would certainly be nice to build an interface on top of one of these models that would accept a list input of data (e.g driver name, circuit, team, starting grid… etc), use the rest of the data available in the Kaggle data, like drivers.csv and circuits.csv to pick up the information needed in the model and give back a prediction.
- As mentioned above, if the model could give a percentage chance of the driver winning rather than a yes or no, that would also be a nice feature. In that sense, you could feed the model with all of the data available for the next Grand Prix on all 20 drivers , and it would return a percentage chance of each driver winning.
- Further to the above point, the model currently only predicts whether a driver will win or not. This could also be extended to say the top three positions or even the top ten, returning a percentage chance for each driver for each position.
- The model could interface with the Ergast Formula One Developer API. (https://ergast.com/mrd/) This API provides all of the Formula One data and is kept up-to-date with all the latest results. Rather than having a static model trained on the Kaggle dataset once, a model linked to Ergast could update itself with new data as it became available. The API would be a simple call using the python requests module and then using the json module to parse the data. From there it would be no different to any of the work done above.
- If weather conditions for the day of the race could ever be sourced, that would also be another excellent feature to add to the model. Different weather conditions can drastically affect the outcome of race.

References:

- https://www.kaggle.com/cjgdev/formula-1-race-data-19502017
- https://scikit-learn.org/stable/
- https://www.analyticsvidhya.com/blog/2018/07/formula-1-aws-machine-learning-build-race-strategies-design-cars/
- https://www.analyticsindiamag.com/the-formula-of-using-artificial-intelligence-in-f1-races/
- https://www.sciencedirect.com/science/article/pii/S2210832717301485
- https://ergast.com/mrd/