

JunHao Chen - JC8289**Troy Mei - TM2903**

Objective: To predict the final race position of a driver and the probability of each position

Data Used: <http://ergast.com/mrd/> - Ergast Developer API provided most of our race data.

Sources Referenced: Borrowed ideas from https://medium.com/@timothy_chong/talk-data-to-me-modeling-of-singapore-grand-prix-2019-b7e04355d54f where championship points represented confidence level before the race. We built upon it and used constructor pointers the race as well.

JunHao Chen - Data Acquisition, Logistic Regression, SVM

Data Acquisition by calling Ergast Developer API. Used race data from 2006 - 2019 only.

Due to the nature of the sport and the technological development of Formula 1 cars, it seems bizarre to use old race data from the 1900s as the cars were less advanced and were far slower. We felt like old data could skew our model and objective to predict recent Formula 1 Grand Prix wins.

2006 - 2013: the introduction of the V8 engines and the modern KERS (Kinetic Energy Recovery Systems) unit

2014 - Current: V6 hybrid engines

```
1 from google.colab import drive
2 drive.mount('/content/drive')
```



Call Ergast Developer API and gather data and insert data into an CSV file.

```

1 import requests
2 import json
3 import copy
4
5
6 def convertTimeToInt(time):
7     time = [char for char in time]
8     time[4] = ":"
9     time = "".join(time)
10    (m, s, ms) = time.split(':')
11    result = float(m) * 60 + float(s) + float(ms) * 1/100
12
13    return (str(result))
14
15 #get race results of a specific season
16 def getRaceResultErgast(season,CSV):
17
18
19    raceResults = requests.get('http://ergast.com/api/f1/{}/results.json?limit=1000'.format(season))
20    qualiResults = requests.get('http://ergast.com/api/f1/{}/qualifying.json?limit=1000'.format(season))
21
22    qualiResults = qualiResults.json()
23    results = raceResults.json()
24    raceCount = 0
25    championshipPoints = {}
26    oldConstructorPoints = {}
27    constructorPoints = {}
28    championshipVic = {}
29    championshipPole = {}
30    constructorVic = {}
31    constructorPole = {}
32
33
34    for i in results['MRData']['RaceTable']['Races']:
35        raceDict = {}
36        champWin = 0
37        qualiWin = 0
38
39        #Get max 1st position from each race

```

```

39 #for quali results for each race
40 for j in qualiResults['MRData']['RaceTable']['Races'][raceCount]['QualifyingResults']:
41     name = j['Driver']['givenName'] + ' ' + j['Driver']['familyName']
42
43     if qualiWin == 0:
44         if name not in championshipPole:
45             championshipPole[name] = 1
46         else:
47             championshipPole[name] += 1
48
49         if j['Constructor']['name'] not in constructorPole:
50             constructorPole[ j['Constructor']['name'] ] = 1
51         else:
52             constructorPole[ j['Constructor']['name'] ] += 1
53
54     try:
55         q1 = ( j['Q1'] )
56         q1 = convertTimeToInt(q1)
57     except:
58         q1 = '0:00:000'
59         q1 = convertTimeToInt(q1)
60     try:
61         q2 = ( j['Q2'] )
62         q2 = convertTimeToInt(q2)
63     except:
64         q2 = q1
65     try:
66         q3 = ( j['Q3'] )
67         q3 = convertTimeToInt(q3)
68     except:
69         q3 = q2
70     quali = [q1,q2,q3]
71     raceDict[name] = quali
72     qualiWin += 1
73
74     #get constructor standing before each race
75
76 for j in i['Results']:
77     # print(j)
78     # print('position: ' + str(j['position']) + ' driverId: ' + str(j['DriverId']) + ' Driver: ' + j['DriverName'] + ' driverFamilyName: ' + j['DriverFamilyName'] + ' Constructor: ' + j['ConstructorName'])

```

```

78 # print(j['position'] + j['grid'] + j['Driver'] + j['givenName'] + j['Driver'] + j['familyName'] + j['Constructor']
79 name = j['Driver']['givenName'] + ' ' + j['Driver']['familyName']
80 constructorName = j['Constructor']['name']
81
82 if champWin == 0:
83     if name not in championshipVic:
84         championshipVic[name] = 1
85     else:
86         championshipVic[name] += 1
87
88     if constructorName not in constructorVic:
89         constructorVic[ constructorName ] = 1
90     else:
91         constructorVic[ constructorName ] += 1
92
93 row = []
94 row.append(str(season))
95 row.append(i['raceName'])
96 row.append(j['position'])
97 row.append(j['grid'])
98 try:
99     quali = raceDict[name]
100     row.append(quali[0])
101     row.append(quali[1])
102     row.append(quali[2])
103 except:
104     row.append('0')
105     row.append('0')
106     row.append('0')
107 row.append(name)
108 row.append(j['Constructor']['name'])
109
110
111 # no points for first race
112 if raceCount == 0:
113     row.append('0')
114     row.append('0')
115     row.append('0')
116     row.append('0')
117     row.append('0')

```

```

117         row.append( 0 )
118         row.append( '0' )
119     else:
120         try:
121             row.append(str(championshipPoints[name]))
122         except:
123             row.append( '0' )
124         try:
125             row.append(str(championshipVic[name]))
126         except:
127             row.append( '0' )
128         try:
129             row.append(str(championshipPole[name]))
130         except:
131             row.append( '0' )
132         try:
133             row.append(str(oldConstructorPoints[constructorName]))
134         except:
135             row.append( '0' )
136         try:
137             row.append(str(constructorVic[constructorName]))
138         except:
139             row.append( '0' )
140         try:
141             row.append(str(constructorPole[constructorName]))
142         except:
143             row.append( '0' )
144
145     if name not in championshipPoints:
146         championshipPoints[name] = float(j['points'])
147     else:
148         championshipPoints[name] += float(j['points'])
149
150     if constructorName not in constructorPoints:
151         constructorPoints[constructorName] = float(j['points'])
152     else:
153         constructorPoints[constructorName] += float(j['points'])
154
155
156     row = ' '.join(row)

```

```

156         row = , .join(row),
157         print(row, file = CSV)
158         champWin +=1
159         raceCount+=1
160         oldConstructorPoints = copy.deepcopy(constructorPoints)
161
162 #get desired results and parse it into CSV
163 def getResults():
164     resultCSV = open('drive/My Drive/results.csv', 'w')
165     print("Year,Race,Final Position,Qualifying Position,Q1,Q2,Q3,Name,Team, Championship Points, Race Vic, Race P
166
167
168     for i in range(2006,2019):
169         getRaceResultErgast(str(i),resultCSV)
170
171     resultCSV.close()
172
173 getResults()
174

```

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib
4 import matplotlib.pyplot as plt
5 %matplotlib inline
6 from sklearn import linear_model, preprocessing
7 from sklearn.preprocessing import LabelEncoder
8 from sklearn.model_selection import KFold
9 from sklearn.metrics import precision_recall_fscore_support
10 import warnings
11 warnings.filterwarnings("ignore")
12 from sklearn.linear_model import LogisticRegression
13 from sklearn import metrics
14 from sklearn.model_selection import train_test_split
15 from sklearn.metrics import confusion_matrix
16 import seaborn as sn
17 import pandas as pd
18 from sklearn import svm

```

You have to manually convert the CSV file into Excel format and upload onto drive

Load Excel file into dataframe

I've shared the Excel formatted version at <https://drive.google.com/file/d/1WwxQ8t22FjyL-IElfrMvCqM8kyIVKsRr/view?usp=sharing> for access

```
1 df = pd.read_excel("drive/My Drive/results.xlsx")
2 df.head()
```

↗

	Year	Race	Final Position	Qualifying Position	Q1	Q2	Q3	Name	Team	Championship Points	Race Vic	Race Pole	Construct Poin
0	2006	Bahrain Grand Prix	1	4	92.32	91.31	91.31	Fernando Alonso	Renault	0.0	0	0	(
1	2006	Bahrain Grand Prix	2	1	93.33	92.32	91.31	Michael Schumacher	Ferrari	0.0	0	0	(
2	2006	Bahrain Grand Prix	3	22	0.00	0.00	0.00	Kimi Räikkönen	McLaren	0.0	0	0	(
3	2006	Bahrain Grand Prix	4	3	92.32	92.32	91.31	Jenson Button	Honda	0.0	0	0	(

Transforming categorical data into numbered labels. EX: Race, Team, Name

```
1 #generating labels for names/teams/races
2 races = np.unique(df['Race'])
3 # print(races)
4
5 LE = LabelEncoder()
6 race_labels = LE.fit_transform(df['Race'])
7 race_mappings = {index: label for index, label in enumerate(LE.classes_)}
8 # print(race_mappings)
9 df['RaceLabel'] = race_labels
```

```

10
11 names = np.unique(df['Name'])
12 name_labels = LE.fit_transform(df['Name'])
13 name_mappings = {index: label for index, label in enumerate(LE.classes_)}
14 # print(name_mappings)
15 df['NameLabel'] = name_labels
16
17 teams = np.unique(df['Team'])
18 team_labels = LE.fit_transform(df['Team'])
19 team_mappings = {index: label for index, label in enumerate(LE.classes_)}
20 # print(team_mappings)
21 df['TeamLabel'] = team_labels
22
23 df.head()

```

↗

	Year	Race	Final Position	Qualifying Position	Q1	Q2	Q3	Name	Team	Championship Points	Race Vic	Race Pole	Construct Poin
0	2006	Bahrain Grand Prix	1	4	92.32	91.31	91.31	Fernando Alonso	Renault	0.0	0	0	(
1	2006	Bahrain Grand Prix	2	1	93.33	92.32	91.31	Michael Schumacher	Ferrari	0.0	0	0	(
2	2006	Bahrain Grand Prix	3	22	0.00	0.00	0.00	Kimi Räikkönen	McLaren	0.0	0	0	(
3	2006	Bahrain Grand Prix	4	3	92.32	92.32	91.31	Jenson Button	Honda	0.0	0	0	(

Target Feature = Final Position

Features = Everything else

```

1 y = np.unique( df['Final Position'].values,return_inverse=True)[1]
2 x = df[['Year','Qualifying Position','Q1','Q2','Q3','Championship Points','Race Vic', 'Race Pole','Constructor
3 x = x.values

```


Initial Accuracy with just tran_test_split

```

1 x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=0)
2
3 logreg = linear_model.LogisticRegression(C=1)
4 logreg.fit(x_train,y_train)
5
6 yhat = logreg.predict(x_test)
7 accuracy = np.mean(yhat == y_test)
8 print("Accuracy of training = {}".format(accuracy))

```

☞ Accuracy of training = 0.12943372744243933

Calculate mean accuracy and standard error with KFold cross validation

```

1 nfold = 10
2 kf = KFold(n_splits=nfold, shuffle=True)
3 acc = []
4
5 for ifold, Ind in enumerate(kf.split(x)):
6
7     # Get training and test data
8     Itr, Its = Ind
9     Xtr = x[Itr,:]
10    ytr = y[Itr]
11    Xts = x[Its,:]
12    yts = y[Its]
13
14    # Fit a model
15    logreg.fit(Xtr, ytr)
16    yhat = logreg.predict(Xts)
17
18    # Measure performance
19    acc.append(np.mean(yhat == yts))
20
21

```

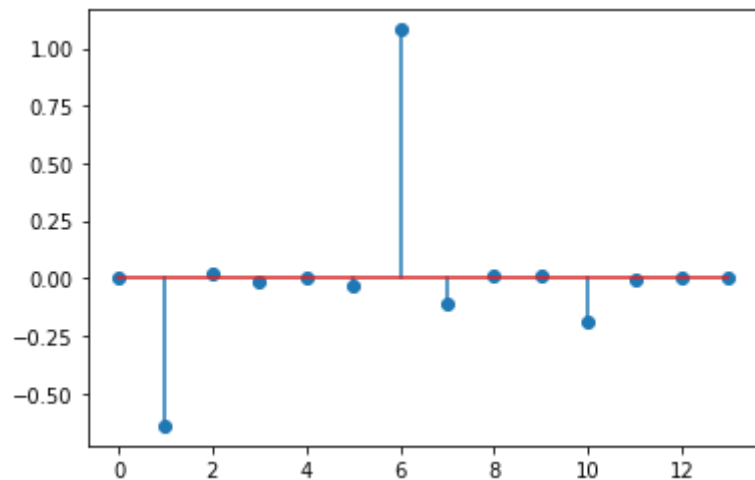
```
22 SE = np.std(acc) / np.sqrt(nfold)
23 print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
```

```
↳ Mean Accuracy rate = 0.12605454038220115 , SE = 0.003512850389355727
```

Weights of each feature from the logistic regression

```
1 logreg.fit(x,y)
2 W = logreg.coef_
3 # print(W)
4 plt.stem(W[0,:])
5
```

```
↳ <StemContainer object of 3 artists>
```



Features = Year, Qualifying Position, Q1, Q2, Q3 ,Championship Points, Race Vic, Race Pole,Constructor Points, Team Vic, Team Pole , Race Vic, NameLabel ,TeamLabel

We see that Qualifying Position has interstingly negative weight.

Race victory has the strongest correlation with predicting final position which makes sense because Formula 1 wins are heavily dominated by the car and less by the skills of the driver.

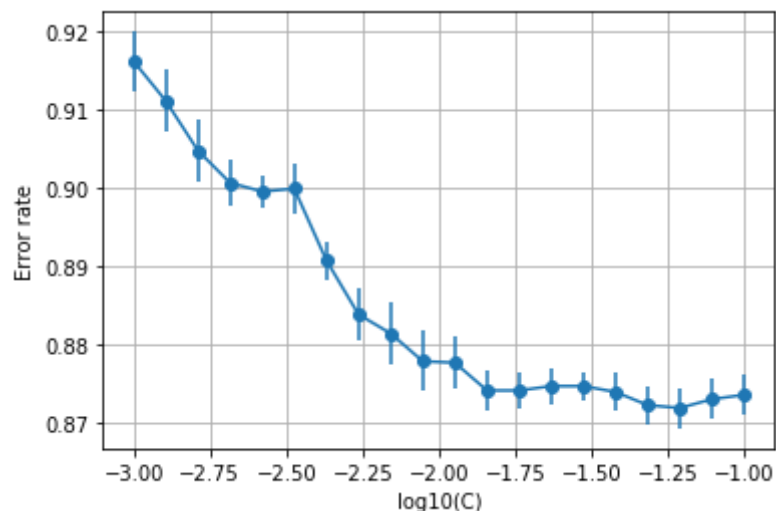
Now we try to apply L1 regularization to make the weights sparse by penalizing 0's

We're also optimizing L1 by finding an optimal C

```
1 npen = 20
2 C_test = np.logspace(-3,-1,npen)
3
4 # Create the cross-validation object and error rate matrix
5 nfold = 10
6 kf = KFold(n_splits=nfold,shuffle=True)
7 err_rate = np.zeros((npen,nfold))
8 num_nonzerocoef = np.zeros((npen,nfold))
9 # Create the logistic regression object
10 logreg = linear_model.LogisticRegression(penalty='l1',warm_start=True)
11
12 # Loop over the folds in the cross-validation
13 for ifold, Ind in enumerate(kf.split(x)):
14
15     # Get training and test data
16     Itr, Its = Ind
17     Xtr = x[Itr,:]
18     ytr = y[Itr]
19     Xts = x[Its,:]
20     yts = y[Its]
21
22     # Loop over penalty levels
23     for ipen, c in enumerate(C_test):
24
25         # Set the penalty level
26         logreg.C= c
27
28         # Fit a model on the training data
29         logreg.fit(Xtr, ytr)
30
31         # Predict the labels on the test set.
32         yhat = logreg.predict(Xts)
33
34         # Measure the accuracy
35         err_rate[ipen,ifold] = np.mean(yhat != yts)
```

36

37 `print("Fold %d" % ifold)` `Fold 0``Fold 1``Fold 2``Fold 3``Fold 4``Fold 5``Fold 6``Fold 7``Fold 8``Fold 9``1 err_mean = np.mean(err_rate, axis=1)``2 err_se = np.std(err_rate,axis=1)/np.sqrt(nfold-1)``3 plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)``4 plt.xlabel('log10(C)')``5 plt.ylabel('Error rate')``6 plt.grid()``7 plt.show()``8 imin = np.argmin(err_mean)``9``10 print("The minimum test error rate = {}, SE={}".format(err_mean[imin], err_se[imin]))``11 print("The C value corresponding to minimum error = {}".format(C_test[imin]))`



The minimum test error rate = 0.871896359324871, SE=0.0026109700964270875

The C value corresponding to minimum error = 0.06158482110660261

The optimal inverse regularization strength is 0.0616 so now we construct the model with L1 regularization using the optimal C

```

1 logreg = linear_model.LogisticRegression(C=C_test[imin],penalty='l1')
2 confusion = np.zeros((24,24))
3 nfold = 10
4 kf = KFold(n_splits=nfold, shuffle=True)
5 acc = []
6 # 8 classes so 8 for matrix size
7
8 for ifold, Ind in enumerate(kf.split(x)):
9
10     # Get training and test data
11     Itr, Its = Ind
12     Xtr = x[Itr,:]
13     ytr = y[Itr]
14     Xts = x[Its,:]
15     yts = y[Its]
16
17     # Fit a model
18     logreg.fit(Xtr, ytr)

```

```

19     yhat = logreg.predict(Xts)
20
21     # Measure performance
22     acc.append(np.mean(yhat == yts))
23     confusion += confusion_matrix(yts,yhat)
24
25
26 SE = np.std(acc) / np.sqrt(nfold)
27 tempSum = np.sum(confusion,1)
28 confusion = confusion / tempSum[np.newaxis,:]
29 print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
30
31

```

☞ Mean Accuracy rate = 0.12624145627005162 , SE = 0.00509793928972676

Our mean accuracy slightly improved with optimal L1 regularization from 0.12605454038220115 to 0.12624145627005162

The confusion matrix for the model is presented below. The probability of predicting final position X given actual position X

```

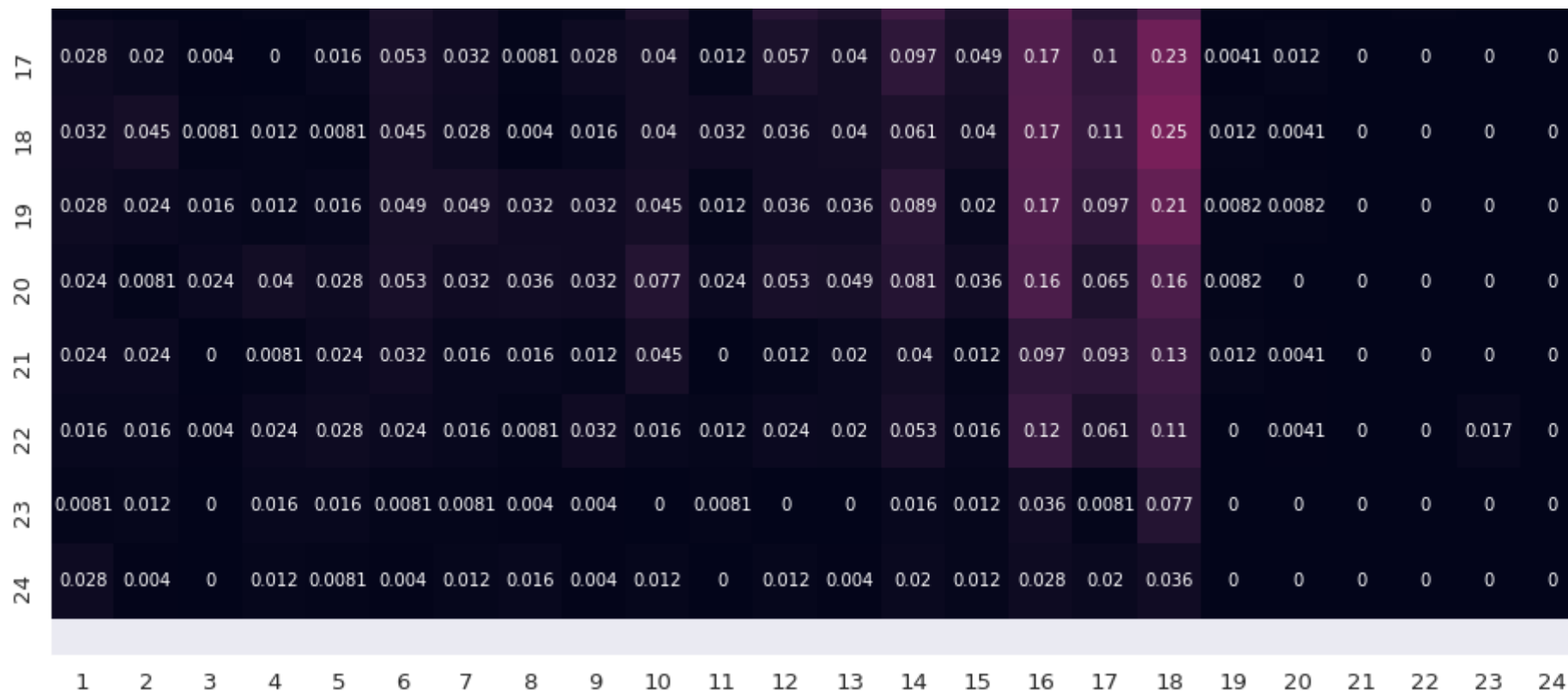
1 df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]],
2                               columns = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]])
3 plt.figure(figsize = (20,20))
4
5 sn.set(font_scale=1.2)#for label size
6 ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
7 bottom, top = ax.get_ylim()
8 ax.set_ylim(bottom + 0.5, top - 0.5)

```

☞

(24.5, -0.5)

1	0.8	0.12	0.02	0.012	0	0.028	0	0	0	0	0	0	0	0	0	0	0	0.016	0	0	0	0	0	0
2	0.32	0.28	0.15	0.069	0.073	0.04	0.012	0.016	0.0081	0.016	0	0	0	0	0.004	0	0	0	0	0.0041	0	0.0066	0	0
3	0.17	0.27	0.061	0.14	0.1	0.089	0.057	0.016	0.0081	0.016	0	0.004	0.0081	0.0081	0	0.004	0.012	0.024	0	0	0	0.0066	0.017	0
4	0.12	0.17	0.11	0.12	0.1	0.14	0.045	0.028	0.024	0.04	0.004	0.0081	0.012	0.004	0	0.012	0	0.024	0	0.025	0	0	0.034	0
5	0.053	0.13	0.093	0.13	0.11	0.17	0.085	0.036	0.069	0.045	0.004	0.0081	0.0081	0	0	0.0081	0.0081	0.016	0.0041	0.0041	0.0065	0.013	0	0
6	0.028	0.069	0.053	0.089	0.13	0.19	0.089	0.049	0.045	0.093	0.016	0.036	0.036	0.016	0.012	0	0.012	0.028	0	0.012	0	0	0	0
7	0.036	0.049	0.032	0.069	0.11	0.12	0.13	0.065	0.053	0.1	0.016	0.032	0.032	0.02	0.02	0.028	0.02	0.04	0.02	0.0082	0	0	0	0
8	0.02	0.045	0.0081	0.053	0.065	0.13	0.16	0.024	0.061	0.13	0.016	0.036	0.045	0.049	0.0081	0.061	0.0081	0.053	0.0082	0.016	0	0	0	0
9	0.024	0.012	0.004	0.028	0.065	0.11	0.11	0.053	0.093	0.13	0.032	0.069	0.049	0.036	0.028	0.04	0.016	0.081	0.012	0.0082	0	0	0	0
10	0.016	0.02	0.004	0.02	0.032	0.085	0.097	0.065	0.065	0.1	0.028	0.081	0.04	0.069	0.02	0.089	0.036	0.097	0.0082	0.02	0	0.0066	0	0
11	0	0.004	0.004	0.012	0.024	0.065	0.11	0.024	0.057	0.11	0.036	0.081	0.089	0.085	0.073	0.057	0.053	0.073	0.025	0.02	0	0	0	0
12	0.012	0.0081	0.004	0.004	0.004	0.053	0.069	0.028	0.045	0.093	0.053	0.057	0.049	0.13	0.093	0.16	0.04	0.081	0.012	0.012	0	0	0	0
13	0.016	0	0	0.0081	0.016	0.036	0.061	0.012	0.053	0.085	0.016	0.073	0.053	0.093	0.077	0.17	0.085	0.14	0.0082	0.0041	0	0	0	0
14	0.012	0.02	0.004	0	0.004	0.049	0.049	0.0081	0.032	0.073	0.012	0.065	0.049	0.11	0.065	0.2	0.053	0.18	0	0.016	0	0	0	0
15	0.004	0.012	0.004	0.024	0.012	0.049	0.045	0.028	0.016	0.049	0.016	0.053	0.045	0.081	0.032	0.23	0.13	0.15	0.012	0.012	0	0	0	0
16	0.0081	0	0	0.012	0.0081	0.057	0.032	0.0081	0.012	0.061	0.012	0.085	0.065	0.14	0.061	0.18	0.081	0.16	0.0082	0.0041	0	0.0066	0	0



Based on the Confusion Matrix, we see that it does pretty well predicting 1st place, but struggles with the rest of positions due to the unpredictability in the mid-field teams. Many things race can occur in a race so it is no surprise that the accuracy is quite low.

What's suprising about this is that it does pretty poorly predicting second place

Comparsion of weights without L1 and with L1 regularixation

```

1 W_l1 = logreg.coef_
2 plt.figure(figsize=(7,7))
3 plt.subplot(2,1,1)
4 # plt.ylim((-0.75,0.05))
5 plt.subplots_adjust(hspace=.5)
6 plt.stem(W[0,:])
7 plt.title('No regularization')

```



```

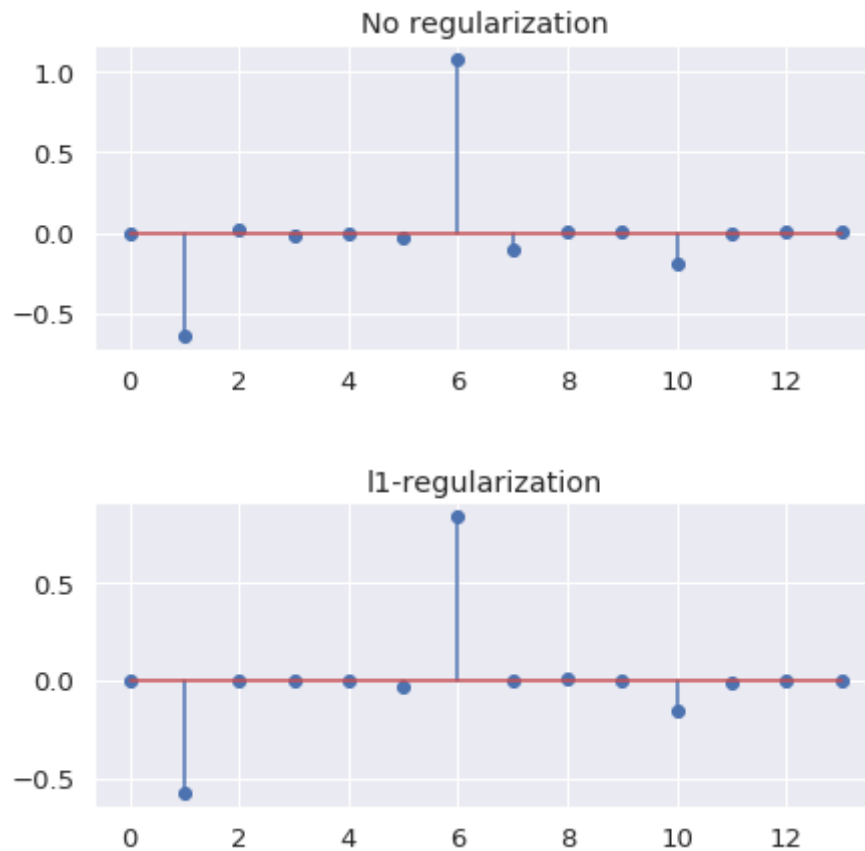
8 plt.subplot(2,1,2)
9 # plt.ylim((-0.75,0.05))
10 plt.stem(W_l1[0,:])
11 plt.title('l1-regularization')

```

```

☐ Text(0.5, 1.0, 'l1-regularization')

```



Now we will try a multi-class logistic regression

```

1 logreg = linear_model.LogisticRegression(solver='newton-cg', multi_class='multinomial')
2 confusion = np.zeros((24,24))
3 nfold = 10
4 kf = KFold(n_splits=nfold, shuffle=True)
5 acc = []
6 # 8 classes so 8 for matrix size

```

```

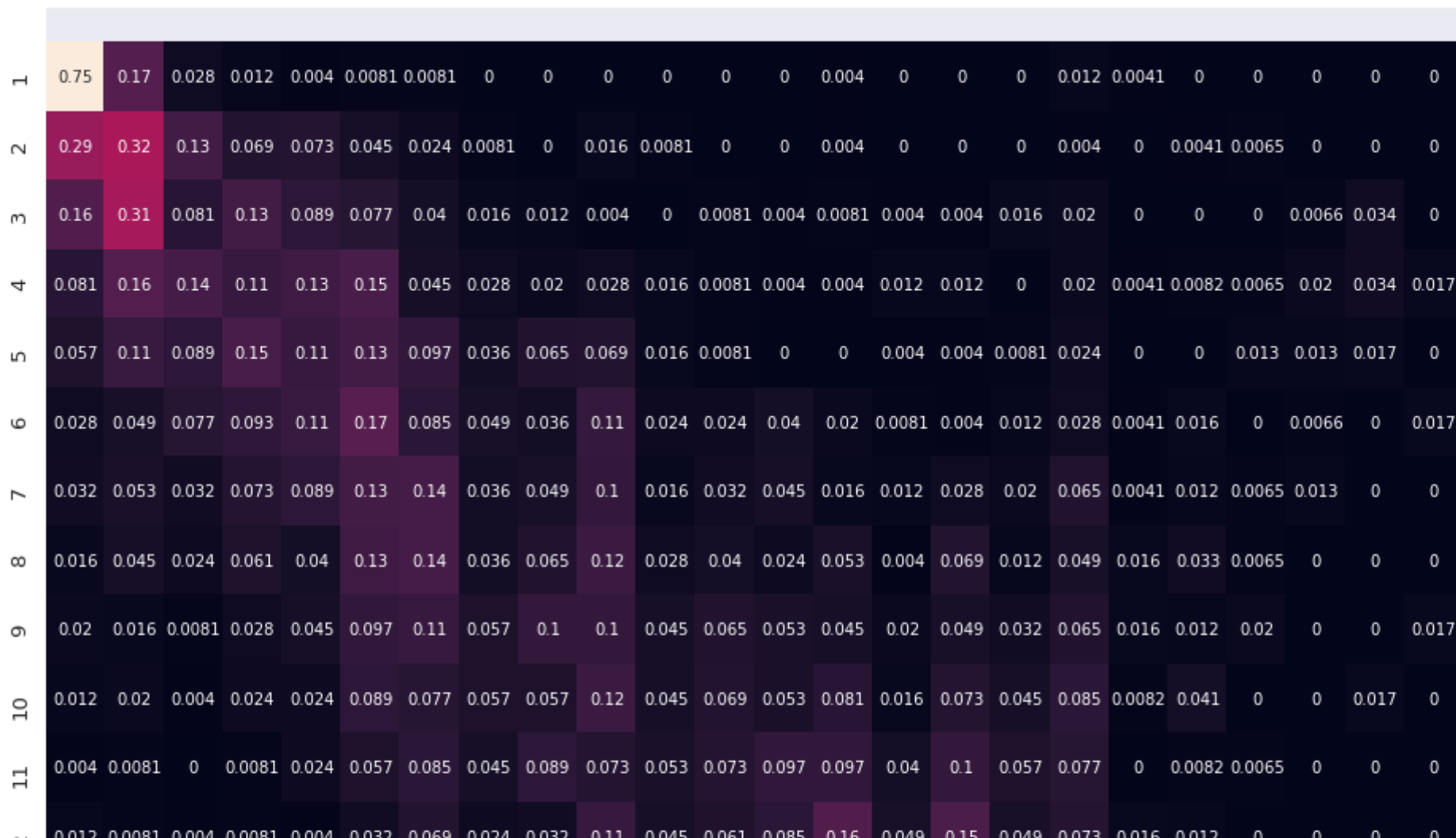
7
8 for ifold, Ind in enumerate(kf.split(x)):
9
10     # Get training and test data
11     Itr, Its = Ind
12     Xtr = x[Itr,:]
13     ytr = y[Itr]
14     Xts = x[Its,:]
15     yts = y[Its]
16
17     # Fit a model
18     logreg.fit(Xtr, ytr)
19     yhat = logreg.predict(Xts)
20
21     # Measure performance
22     acc.append(np.mean(yhat == yts))
23     confusion += confusion_matrix(yts,yhat)
24
25     print("Fold %d" % ifold)
26
27
28 SE = np.std(acc) / np.sqrt(nfold)
29 tempSum = np.sum(confusion,1)
30 confusion = confusion / tempSum[np.newaxis,:]
31 print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
32
33 df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]],
34                        columns = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]])
35 plt.figure(figsize = (20,20))
36
37 sn.set(font_scale=1.2)#for label size
38 ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
39 bottom, top = ax.get_ylim()
40 ax.set_ylim(bottom + 0.5, top - 0.5)

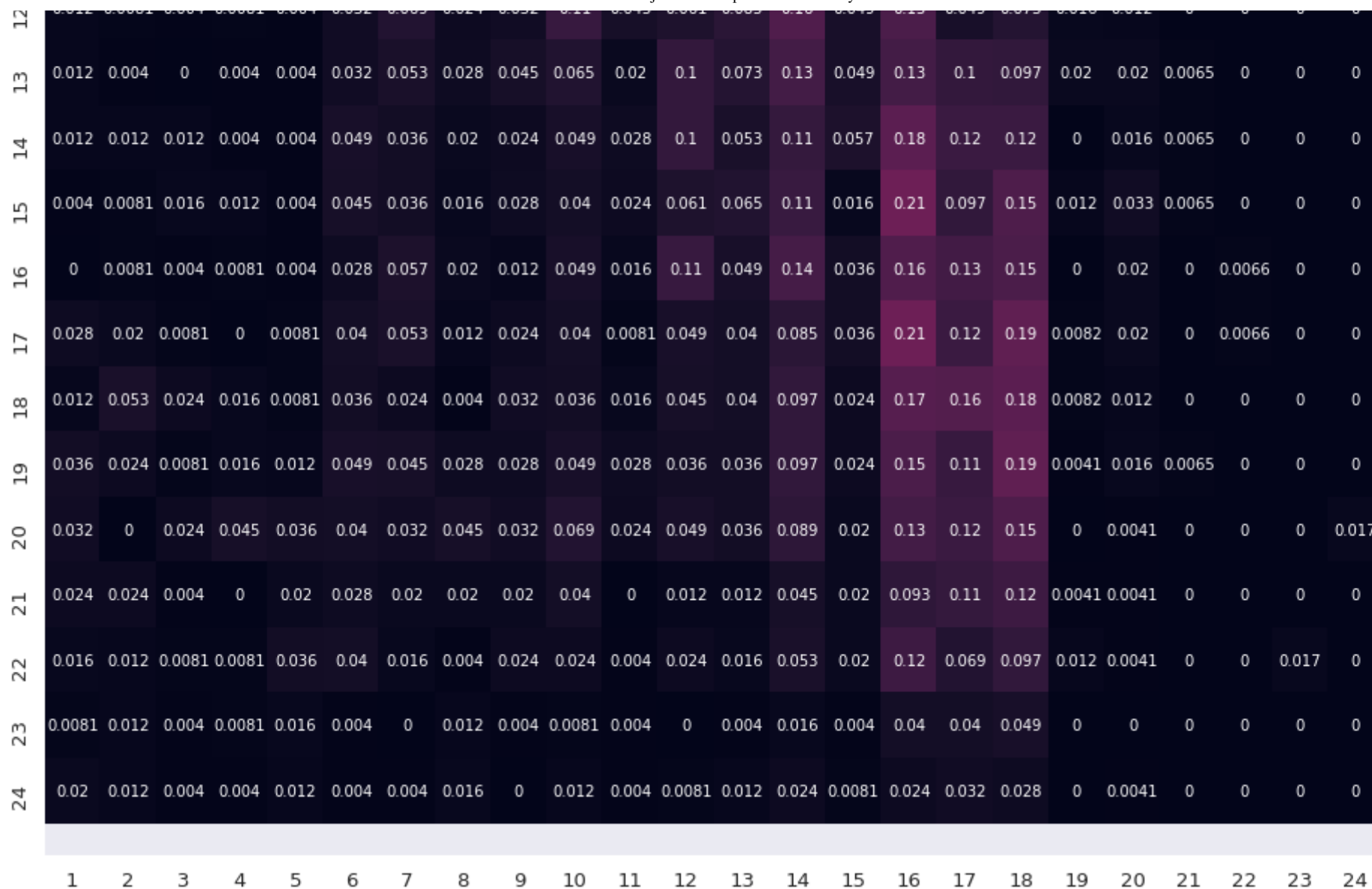
```



Fold 0
 Fold 1
 Fold 2
 Fold 3
 Fold 4
 Fold 5
 Fold 6
 Fold 7
 Fold 8
 Fold 9

Mean Accuracy rate = 0.12492676802901378 , SE = 0.0025905347217149597
 (24.5, -0.5)





Mean accuracy using multi-class model performed slightly worse with 0.12492676802901378 compared to logistic regression of 0.12624145627005162

Now we try to optimize C for multi-class model with L2 regularization since multi-class does not offer L1

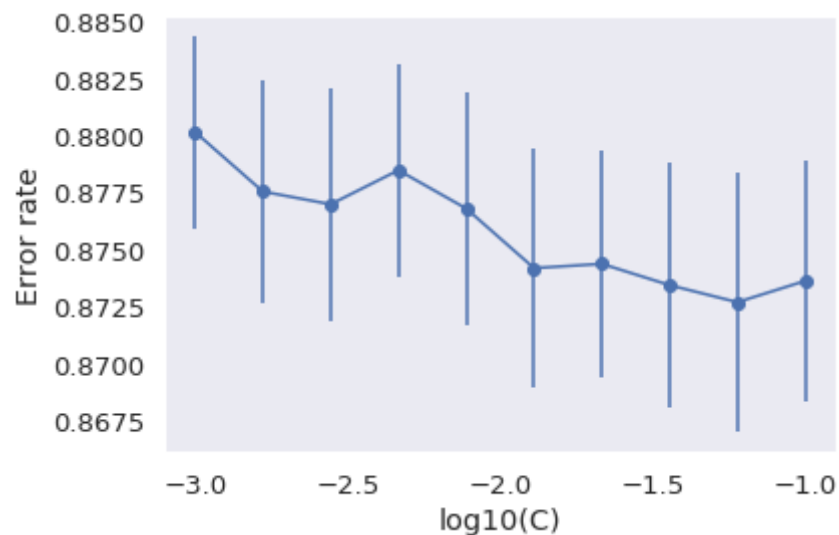
```
1 npen = 10
2 C_test = np.logspace(-3,-1,npen)
3 # Create the cross-validation object and error rate matrix
4 nfold = 10
5 kf = KFold(n_splits=nfold,shuffle=True)
6 err_rate = np.zeros((npen,nfold))
7 num_nonzerocoef = np.zeros((npen,nfold))
8 # Create the logistic regression object
9 logreg = linear_model.LogisticRegression(solver='newton-cg', multi_class='multinomial', penalty='l2')
10 # Loop over the folds in the cross-validation
11 for ifold, Ind in enumerate(kf.split(x)):
12
13     # Get training and test data
14     Itr, Its = Ind
15     Xtr = x[Itr,:]
16     ytr = y[Itr]
17     Xts = x[Its,:]
18     yts = y[Its]
19
20     # Loop over penalty levels
21     for ipen, c in enumerate(C_test):
22
23         # Set the penalty level
24         logreg.C= c
25
26         # Fit a model on the training data
27         logreg.fit(Xtr, ytr)
28
29         # Predict the labels on the test set.
30         yhat = logreg.predict(Xts)
31
32         # Measure the accuracy
33         err_rate[ipen,ifold] = np.mean(yhat != yts)
34     print("Fold %d" % ifold)
```



Fold 0
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Fold 6
Fold 7
Fold 8
Fold 9

```
1 err_mean = np.mean(err_rate, axis=1)
2 err_se = np.std(err_rate,axis=1)/np.sqrt(nfold-1)
3 plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
4 plt.xlabel('log10(C)')
5 plt.ylabel('Error rate')
6 plt.grid()
7 plt.show()
8 imin = np.argmin(err_mean)
9
10 print("The minimum test error rate = {}, SE={}".format(err_mean[imin], err_se[imin]))
11 print("The C value corresponding to minimum error = {}".format(C_test[imin]))
```





The minimum test error rate = 0.8726429767052588, SE=0.005711257265191925

The C value corresponding to minimum error = 0.05994842503189409

```

1 logreg = linear_model.LogisticRegression(C=C_test[imin], solver='newton-cg', multi_class='multinomial', penalty
2 confusion = np.zeros((24,24))
3 nfold = 10
4 kf = KFold(n_splits=nfold, shuffle=True)
5 acc = []
6 # 8 classes so 8 for matrix size
7
8 for ifold, Ind in enumerate(kf.split(x)):
9
10     # Get training and test data
11     Itr, Its = Ind
12     Xtr = x[Itr,:]
13     ytr = y[Itr]
14     Xts = x[Its,:]
15     yts = y[Its]
16
17     # Fit a model
18     logreg.fit(Xtr, ytr)
19     yhat = logreg.predict(Xts)
20

```

```
21 # Measure performance
22 acc.append(np.mean(yhat == yts))
23 confusion += confusion_matrix(yts,yhat)
24
25
26 SE = np.std(acc) / np.sqrt(nfold)
27 tempSum = np.sum(confusion,1)
28 confusion = confusion / tempSum[np.newaxis,:]
29 print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
```

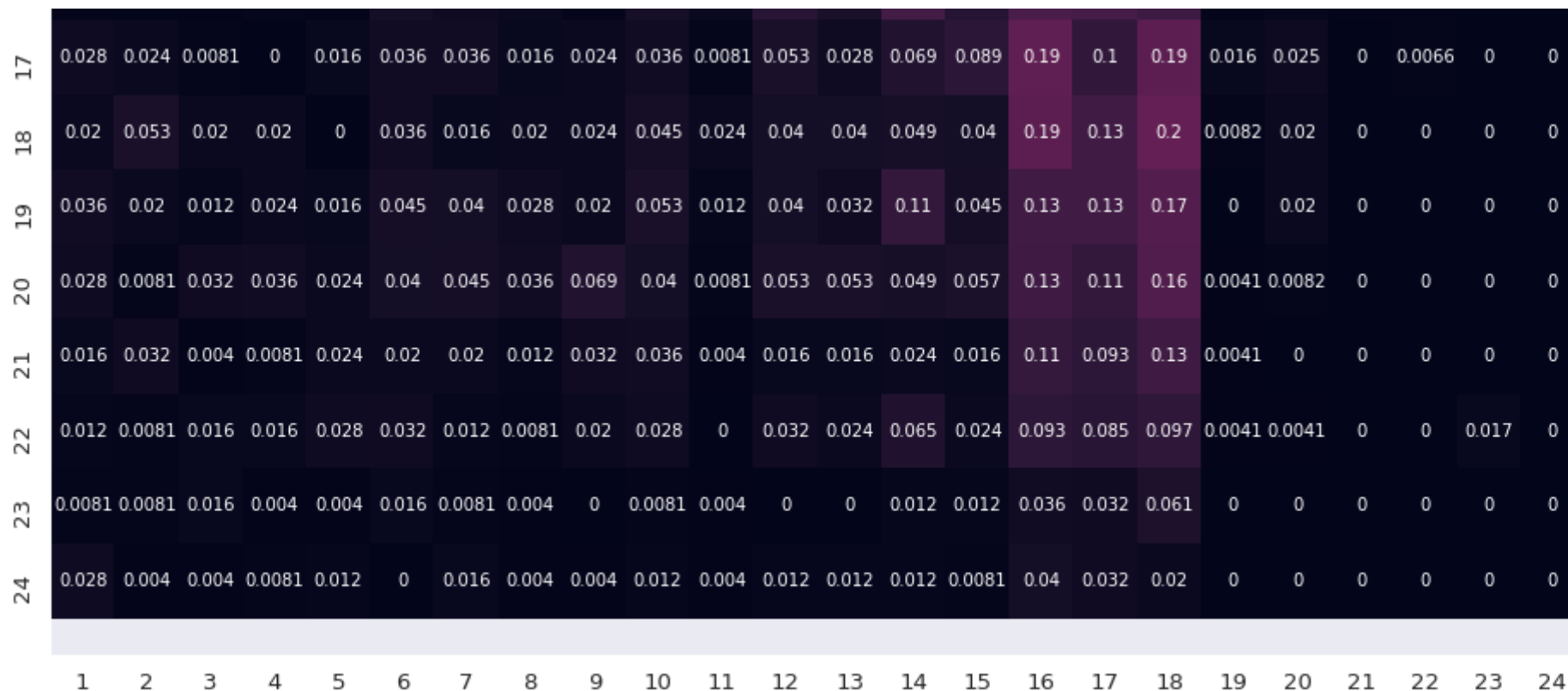
☞ Mean Accuracy rate = 0.12174815176454179 , SE = 0.003204075834675082

```
1 df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]],
2                               columns = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]])
3 plt.figure(figsize = (20,20))
4
5 sn.set(font_scale=1.2)#for label size
6 ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
7 bottom, top = ax.get_ylim()
8 ax.set_ylim(bottom + 0.5, top - 0.5)
```

☞

(24.5, -0.5)

1	0.78	0.14	0.028	0.016	0	0.016	0.004	0	0	0	0	0	0	0.004	0	0	0.004	0.0081	0	0	0	0	0	0
2	0.28	0.33	0.13	0.093	0.065	0.028	0.032	0.004	0.0081	0.02	0	0	0	0	0	0.004	0	0.004	0	0	0	0.0066	0	0
3	0.15	0.32	0.073	0.13	0.073	0.089	0.061	0.012	0.004	0.012	0	0.004	0.004	0.0081	0.004	0.012	0.0081	0.024	0	0	0	0	0.034	0
4	0.081	0.17	0.13	0.14	0.13	0.11	0.053	0.036	0.016	0.036	0.0081	0.016	0.004	0	0.004	0.012	0	0.016	0	0.02	0.0065	0.0066	0.034	0
5	0.057	0.13	0.11	0.11	0.093	0.15	0.089	0.045	0.077	0.036	0.004	0.004	0.0081	0.012	0.004	0	0.012	0.02	0.0041	0	0.013	0.02	0	0
6	0.024	0.045	0.081	0.073	0.13	0.16	0.089	0.057	0.069	0.085	0.012	0.032	0.032	0.02	0.0081	0	0.024	0.028	0.012	0.016	0	0.0066	0	0
7	0.032	0.049	0.02	0.085	0.11	0.13	0.14	0.024	0.069	0.097	0.028	0.012	0.045	0.028	0.02	0.016	0.02	0.045	0.016	0.0082	0.0065	0.0066	0	0
8	0.016	0.032	0.024	0.049	0.057	0.11	0.15	0.045	0.065	0.13	0.024	0.057	0.024	0.057	0.012	0.053	0.012	0.045	0.0041	0.037	0.0065	0	0	0
9	0.028	0.016	0.004	0.024	0.045	0.093	0.12	0.061	0.077	0.15	0.032	0.049	0.036	0.032	0.053	0.049	0.032	0.061	0.016	0.0041	0.013	0	0	0.017
10	0.012	0.024	0.0081	0.02	0.012	0.093	0.097	0.073	0.065	0.097	0.016	0.073	0.061	0.081	0.032	0.061	0.045	0.077	0.02	0.033	0	0	0	0
11	0.004	0.0081	0.004	0.0081	0.012	0.069	0.089	0.04	0.077	0.093	0.016	0.089	0.069	0.12	0.069	0.053	0.053	0.085	0.012	0.025	0	0	0.017	0
12	0.012	0.0081	0.004	0.0081	0.004	0.028	0.077	0.045	0.049	0.085	0.04	0.04	0.053	0.15	0.085	0.16	0.049	0.089	0.0082	0.0082	0	0	0	0
13	0.016	0	0.004	0	0.012	0.028	0.065	0.02	0.049	0.061	0.012	0.11	0.065	0.12	0.077	0.13	0.089	0.13	0.012	0.0082	0	0	0	0
14	0.012	0.024	0.012	0	0.004	0.049	0.032	0.02	0.02	0.061	0.032	0.097	0.045	0.089	0.081	0.19	0.085	0.13	0	0.016	0	0	0	0
15	0	0.016	0.012	0.004	0.016	0.045	0.024	0.028	0.032	0.04	0.024	0.053	0.053	0.089	0.036	0.21	0.15	0.14	0.016	0.016	0	0	0	0
16	0.0081	0	0.004	0.012	0.004	0.04	0.032	0.024	0.0081	0.049	0.02	0.085	0.053	0.14	0.077	0.15	0.14	0.13	0.0082	0.012	0.0065	0.0066	0	0



Trying out SVM Model

```

1 from sklearn import svm
2 svc = svm.SVC(probability=False, kernel="rbf", C=2.8, gamma=.0073)
3
4 nfold = 10
5 kf = KFold(n_splits=nfold, shuffle=True)
6 acc = []
7 # 8 classes so 8 for matrix size
8
9 for ifold, Ind in enumerate(kf.split(x)):
10
11     # Get training and test data
12     Itr, Its = Ind
13     xtr = x[Itr].T

```

```

13     acc = []
14     ytr = y[Itr]
15     Xts = x[Its,:]
16     yts = y[Its]
17
18     # Fit a model
19     svc.fit(Xtr, ytr)
20     yhat = svc.predict(Xts)
21
22     # Measure performance
23     acc.append(np.mean(yhat == yts))
24
25     print("Fold: %d" %ifold)
26
27
28 SE = np.std(acc) / np.sqrt(nfold)
29 print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))

```

```

↳ Fold: 0
   Fold: 1
   Fold: 2
   Fold: 3
   Fold: 4
   Fold: 5
   Fold: 6
   Fold: 7
   Fold: 8
   Fold: 9
   Mean Accuracy rate = 0.07955049518761334 , SE = 0.002653943187180351

```

SVM seems to perform really bad. I assume it is because the slack from SVM reduces the accuracy of predicting the exact position between the difference between 5th-7th in the mid-field might be minimal.

Conclusion: Logistic Regression with L1 regularization seems to perform the best

```

1 npen = 20
2 C_test = np.logspace(-3,-1,npen)
3

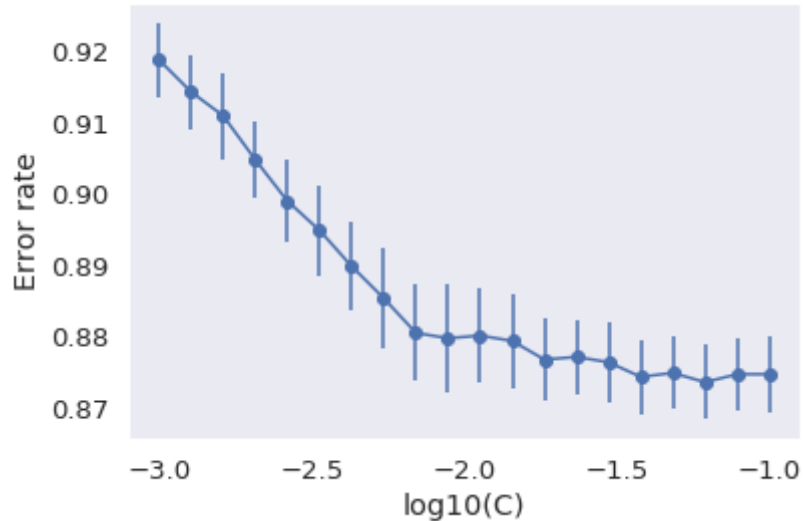
```

```
4 # Create the cross-validation object and error rate matrix
5 nfold = 10
6 kf = KFold(n_splits=nfold,shuffle=True)
7 err_rate = np.zeros((npen,nfold))
8 num_nonzerocoef = np.zeros((npen,nfold))
9 # Create the logistic regression object
10 logreg = linear_model.LogisticRegression(penalty='l1',warm_start=True)
11
12 # Loop over the folds in the cross-validation
13 for ifold, Ind in enumerate(kf.split(x)):
14
15     # Get training and test data
16     Itr, Its = Ind
17     Xtr = x[Itr,:]
18     ytr = y[Itr]
19     Xts = x[Its,:]
20     yts = y[Its]
21
22     # Loop over penalty levels
23     for ipen, c in enumerate(C_test):
24
25         # Set the penalty level
26         logreg.C= c
27
28         # Fit a model on the training data
29         logreg.fit(Xtr, ytr)
30
31         # Predict the labels on the test set.
32         yhat = logreg.predict(Xts)
33
34         # Measure the accuracy
35         err_rate[ipen,ifold] = np.mean(yhat != yts)
36
37     print("Fold %d" % ifold)
38
39
40 1 err_mean = np.mean(err_rate, axis=1)
41 2 err_se = np.std(err_rate,axis=1)/np.sqrt(nfold-1)
42 3 plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
```

```

4 plt.xlabel('log10(C)')
5 plt.ylabel('Error rate')
6 plt.grid()
7 plt.show()
8 imin = np.argmin(err_mean)
9
10 print("The minimum test error rate = {}, SE={}".format(err_mean[imin], err_se[imin]))
11 print("The C value corresponding to minimum error = {}".format(C_test[imin]))

```



The minimum test error rate = 0.8735799972102107, SE=0.0052124396161550264
 The C value corresponding to minimum error = 0.06158482110660261

```

1 logreg = linear_model.LogisticRegression(C=C_test[imin],penalty='l1')
2 confusion = np.zeros((24,24))
3 nfold = 10
4 kf = KFold(n_splits=nfold, shuffle=True)
5 acc = []
6
7 for ifold, Ind in enumerate(kf.split(x)):
8
9     # Get training and test data
10     Itr, Its = Ind
11     Xtr = x[Itr,:]
12     ytr = y[Itr]

```

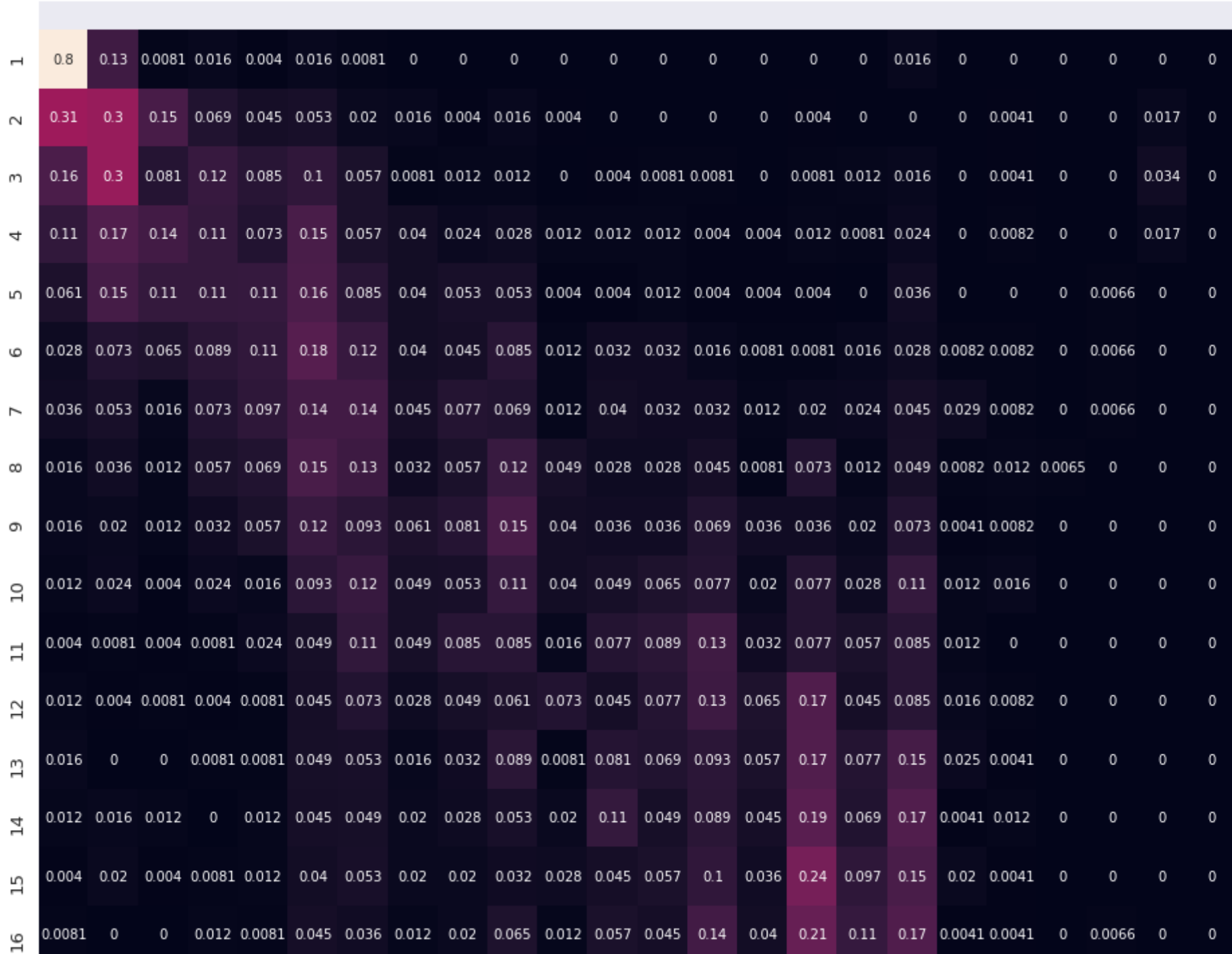
```
13     Xts = x[Its,:]
14     yts = y[Its]
15
16     # Fit a model
17     logreg.fit(Xtr, ytr)
18     yhat = logreg.predict(Xts)
19
20     # Measure performance
21     acc.append(np.mean(yhat == yts))
22     confusion += confusion_matrix(yts,yhat)
23
24
25 SE = np.std(acc) / np.sqrt(nfold)
26 tempSum = np.sum(confusion,1)
27 confusion = confusion / tempSum[np.newaxis,:]
28 print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
```

```
☞ Mean Accuracy rate = 0.12755126237969033 , SE = 0.0038867045624915077
```

```
1 df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]],
2                        columns = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]])
3 plt.figure(figsize = (20,20))
4
5 sn.set(font_scale=1.2)#for label size
6 ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
7 bottom, top = ax.get_ylim()
8 ax.set_ylim(bottom + 0.5, top - 0.5)
```

```
☞
```

(24.5, -0.5)



17	0.032	0.024	0	0	0.024	0.04	0.04	0.004	0.032	0.028	0.016	0.036	0.028	0.1	0.04	0.23	0.1	0.2	0.012	0.0082	0	0	0	0
18	0.028	0.049	0.004	0.012	0	0.069	0.036	0.004	0.012	0.036	0.024	0.016	0.061	0.065	0.036	0.19	0.093	0.25	0.012	0.0041	0	0	0	0
19	0.032	0.024	0.012	0.024	0.024	0.032	0.036	0.045	0.036	0.049	0.024	0.024	0.036	0.085	0.036	0.16	0.11	0.19	0.0041	0.0041	0	0	0	0
20	0.024	0.012	0.02	0.036	0.032	0.045	0.053	0.028	0.049	0.057	0.02	0.04	0.036	0.1	0.028	0.14	0.061	0.19	0.0082	0	0	0	0	0
21	0.028	0.02	0.0081	0.012	0.016	0.02	0.024	0.012	0.028	0.028	0.016	0.016	0.004	0.036	0.0081	0.12	0.089	0.12	0.0041	0.0041	0	0	0	0
22	0.012	0.016	0.012	0.012	0.028	0.032	0.016	0.0081	0.016	0.04	0.004	0.024	0.024	0.045	0.016	0.15	0.04	0.11	0.0041	0	0	0	0.017	0
23	0.0081	0.016	0.004	0.0081	0.0081	0.012	0.012	0.004	0.004	0.004	0	0.004	0.004	0.012	0	0.049	0.024	0.061	0	0	0	0	0	0
24	0.024	0.0081	0.004	0.004	0.016	0.004	0.012	0.0081	0	0.016	0.0081	0.016	0.004	0.0081	0.0081	0.036	0.012	0.045	0	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

```

1 print(race_mappings)
2 print(team_mappings)
3 print(name_mappings)

```

```

↳ {0: 'Abu Dhabi Grand Prix', 1: 'Australian Grand Prix', 2: 'Austrian Grand Prix', 3: 'Azerbaijan Grand Prix',
{0: 'BMW Sauber', 1: 'Brawn', 2: 'Caterham', 3: 'Ferrari', 4: 'Force India', 5: 'HRT', 6: 'Haas F1 Team', 7:
{0: 'Adrian Sutil', 1: 'Alexander Rossi', 2: 'Alexander Wurz', 3: 'André Lotterer', 4: 'Anthony Davidson', 5:

```

Features = Year, Qualifying Position, Q1, Q2, Q3 ,Championship Points, Race Vic, Race Pole,Constructor Points, Team Vic, Team Pole , Race Position, NameLabel ,TeamLabel

Predicting 2019 Australian Grand Prix of Lewis Hamilton

Actual: Lewis Hamilton: Q1: 1:22.043 Q2: 1:21.014 Q3: 1:20.486 Pole Position: 1 Final Race Position:2

Double-click (or enter) to edit

```
1 test = [[2019 , 1 , 82.043 ,61.014 ,60.486 ,0 ,0 ,0 ,0, 0 , 1, 40, 14 ]]
2 result = logreg.predict(test)
3 print(result+1)
4
5 proba = logreg.predict_proba(test)
6 print(proba)
```

```
↳ [1]
[[0.16108987 0.11546597 0.08827704 0.0695838 0.07034709 0.08589438
 0.06200464 0.0496314 0.05502098 0.04341341 0.03060956 0.02688846
 0.02460075 0.02165354 0.01267968 0.01156981 0.00859482 0.00977716
 0.01574601 0.01898879 0.00694425 0.00668442 0.00108118 0.00345303]]
```

Predicting 2019 Bahrain Grand Prix of Daniel Ricciardo

Actual: Quali Pos: 10 Q1: 1:29.859 Q2: 1:29.488 Q3: N/A Final Pos: 18

```
1 test = [[2019 ,10 , 89.859 ,89.488 ,0 ,0 ,0 ,0 ,6, 0, 0 , 4, 13, 14 ]]
2 result = logreg.predict(test)
3 print(result+1)
4
5 proba = logreg.predict_proba(test)
6 print(proba)
```

```
↳ [13]
[[0.00209206 0.01309737 0.02605341 0.03642539 0.0006909 0.04420738
 0.010557 0.05208603 0.07066764 0.06351455 0.05710807 0.05740767
 0.1800891 0.06847673 0.04698388 0.04407208 0.02198871 0.04518735
 0.05114299 0.05152163 0.02124341 0.02422029 0.00561502 0.00555134]]
```

Troy Mei - Neutral Network

```
1 import tensorflow as tf
2 import pandas as pd
```

```

3 import numpy as np
4 import matplotlib.pyplot as plt
5 import xlrd
6
7 from tensorflow.keras.layers import BatchNormalization
8 from tensorflow.keras.models import Model, Sequential
9 from tensorflow.keras.layers import Dense, Activation
10 from sklearn.model_selection import train_test_split
11 from tensorflow.keras import optimizers
12 import tensorflow.keras.backend as K

1 data = pd.read_excel("drive/My Drive/results.xlsx")
2
3 df = pd.DataFrame(data, columns=["Year", "Race", "Final Position",
4                                 "Qualifying Position", "Q1", "Q2", "Q3",
5                                 "Name", "Team", "Championship Points",
6                                 "Race Vic", "Race Pole", "Constructor Points",
7                                 "Team Vic", "Team Pole"])
8 #print(df.isnull().any())
9
10 input_data = df[["Qualifying Position", "Q1", "Q2", "Q3",
11                  "Championship Points", "Race Vic",
12                  "Race Pole", "Constructor Points",
13                  "Team Vic", "Team Pole", "Final Position"]]
14
15 n = input_data.shape[0]
16 input_data = input_data.values
17
18 tr_start = 0
19 tr_end = int(np.floor(0.8*n))
20 ts_start = tr_end + 1
21 ts_end = n
22
23 tr_data = input_data[tr_start:tr_end]
24 ts_data = input_data[ts_start:ts_end]
25
26 xtr = tr_data[:,0:-1]
27 ytr = tr_data[:, -1]

```

```

28 xts = ts_data[:,0:-1]
29 yts = ts_data[:, -1]
30
31 xmean = np.mean(xtr,axis=0)
32 xstd = np.std(xtr,axis=0)
33 #print(xmean, xstd)
34 xtr_scale = (xtr-xmean[None,:])/xstd[None,:]
35 xts_scale = (xts-xmean[None,:])/xstd[None,:]
36
37 K.clear_session()
38
39 nin = xtr.shape[1]
40 nout = np.max(ytr)+1
41 nh = round((nin+nout)/2)
42 #nh = 256
43 print(nh, nin, nout)
44 model = Sequential()
45 model.add(Dense(units=nh, input_shape=(nin,), activation="relu", name="hidden"))
46 model.add(BatchNormalization())
47 model.add(Dense(units=nout, activation="softmax", name="output"))
48 model.summary()

```

```

↳ 18.0 10 25.0
Model: "sequential"

```

Layer (type)	Output Shape	Param #
=====		
hidden (Dense)	(None, 18)	198

batch_normalization (BatchNo	(None, 18)	72

output (Dense)	(None, 25)	475
=====		
Total params: 745		
Trainable params: 709		
Non-trainable params: 36		

```

1 opt = optimizers.Adam(lr=0.001)
2 model.compile(optimizer=opt,

```

```
3         loss='sparse_categorical_crossentropy',  
4         metrics=['accuracy'])  
5  
6 hist = model.fit(xtr_scale, ytr, epochs=100, batch_size=100, validation_data=(xts_scale,yts))
```



Train on 4284 samples, validate on 1070 samples

Epoch 1/100

4284/4284 [=====] - 0s 54us/sample - loss: 3.3963 - acc: 0.0565 - val_loss: 3.2054 -

Epoch 2/100

4284/4284 [=====] - 0s 25us/sample - loss: 3.1946 - acc: 0.0745 - val_loss: 3.0867 -

Epoch 3/100

4284/4284 [=====] - 0s 22us/sample - loss: 3.0929 - acc: 0.0962 - val_loss: 3.0139 -

Epoch 4/100

4284/4284 [=====] - 0s 22us/sample - loss: 3.0262 - acc: 0.1097 - val_loss: 2.9612 -

Epoch 5/100

4284/4284 [=====] - 0s 23us/sample - loss: 2.9702 - acc: 0.1190 - val_loss: 2.9180 -

Epoch 6/100

4284/4284 [=====] - 0s 22us/sample - loss: 2.9277 - acc: 0.1256 - val_loss: 2.8789 -

Epoch 7/100

4284/4284 [=====] - 0s 22us/sample - loss: 2.8910 - acc: 0.1235 - val_loss: 2.8492 -

Epoch 8/100

4284/4284 [=====] - 0s 22us/sample - loss: 2.8686 - acc: 0.1237 - val_loss: 2.8248 -

Epoch 9/100

4284/4284 [=====] - 0s 26us/sample - loss: 2.8444 - acc: 0.1342 - val_loss: 2.8031 -

Epoch 10/100

4284/4284 [=====] - 0s 27us/sample - loss: 2.8298 - acc: 0.1347 - val_loss: 2.7915 -

Epoch 11/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.8213 - acc: 0.1305 - val_loss: 2.7803 -

Epoch 12/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.8117 - acc: 0.1349 - val_loss: 2.7693 -

Epoch 13/100

4284/4284 [=====] - 0s 23us/sample - loss: 2.8013 - acc: 0.1394 - val_loss: 2.7626 -

Epoch 14/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7918 - acc: 0.1466 - val_loss: 2.7540 -

Epoch 15/100

4284/4284 [=====] - 0s 25us/sample - loss: 2.7883 - acc: 0.1403 - val_loss: 2.7498 -

Epoch 16/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7778 - acc: 0.1457 - val_loss: 2.7444 -

Epoch 17/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7765 - acc: 0.1415 - val_loss: 2.7412 -

Epoch 18/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7759 - acc: 0.1384 - val_loss: 2.7377 -

Epoch 19/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7675 - acc: 0.1401 - val_loss: 2.7318 -

Epoch 20/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7651 - acc: 0.1417 - val_loss: 2.7290 -

Epoch 21/100

4284/4284 [=====] - 0s 24us/sample - loss: 2.7652 - acc: 0.1422 - val_loss: 2.7239 -

```
Epoch 22/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7575 - acc: 0.1422 - val_loss: 2.7223 -
Epoch 23/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7546 - acc: 0.1405 - val_loss: 2.7203 -
Epoch 24/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7491 - acc: 0.1412 - val_loss: 2.7171 -
Epoch 25/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7476 - acc: 0.1443 - val_loss: 2.7151 -
Epoch 26/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7511 - acc: 0.1450 - val_loss: 2.7111 -
Epoch 27/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7470 - acc: 0.1438 - val_loss: 2.7096 -
Epoch 28/100
4284/4284 [=====] - 0s 27us/sample - loss: 2.7418 - acc: 0.1487 - val_loss: 2.7069 -
Epoch 29/100
4284/4284 [=====] - 0s 26us/sample - loss: 2.7399 - acc: 0.1450 - val_loss: 2.7064 -
Epoch 30/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7358 - acc: 0.1522 - val_loss: 2.7040 -
Epoch 31/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7357 - acc: 0.1513 - val_loss: 2.7026 -
Epoch 32/100
4284/4284 [=====] - 0s 25us/sample - loss: 2.7336 - acc: 0.1494 - val_loss: 2.7008 -
Epoch 33/100
4284/4284 [=====] - 0s 21us/sample - loss: 2.7329 - acc: 0.1468 - val_loss: 2.6992 -
Epoch 34/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7296 - acc: 0.1508 - val_loss: 2.6954 -
Epoch 35/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7264 - acc: 0.1459 - val_loss: 2.6943 -
Epoch 36/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7232 - acc: 0.1499 - val_loss: 2.6926 -
Epoch 37/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7240 - acc: 0.1529 - val_loss: 2.6926 -
Epoch 38/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7283 - acc: 0.1515 - val_loss: 2.6908 -
Epoch 39/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7233 - acc: 0.1529 - val_loss: 2.6885 -
Epoch 40/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7214 - acc: 0.1508 - val_loss: 2.6894 -
Epoch 41/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7175 - acc: 0.1496 - val_loss: 2.6873 -
Epoch 42/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7156 - acc: 0.1541 - val_loss: 2.6887 -
Epoch 43/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7177 - acc: 0.1534 - val_loss: 2.6877 -
```

```
Epoch 44/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7137 - acc: 0.1475 - val_loss: 2.6854 -
Epoch 45/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7146 - acc: 0.1534 - val_loss: 2.6840 -
Epoch 46/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7185 - acc: 0.1557 - val_loss: 2.6858 -
Epoch 47/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7119 - acc: 0.1573 - val_loss: 2.6870 -
Epoch 48/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7136 - acc: 0.1496 - val_loss: 2.6840 -
Epoch 49/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7103 - acc: 0.1555 - val_loss: 2.6833 -
Epoch 50/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7115 - acc: 0.1536 - val_loss: 2.6840 -
Epoch 51/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7128 - acc: 0.1510 - val_loss: 2.6821 -
Epoch 52/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7083 - acc: 0.1529 - val_loss: 2.6805 -
Epoch 53/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7096 - acc: 0.1510 - val_loss: 2.6816 -
Epoch 54/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7056 - acc: 0.1466 - val_loss: 2.6808 -
Epoch 55/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.7109 - acc: 0.1482 - val_loss: 2.6809 -
Epoch 56/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7067 - acc: 0.1529 - val_loss: 2.6816 -
Epoch 57/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7053 - acc: 0.1552 - val_loss: 2.6780 -
Epoch 58/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7065 - acc: 0.1564 - val_loss: 2.6821 -
Epoch 59/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7068 - acc: 0.1562 - val_loss: 2.6822 -
Epoch 60/100
4284/4284 [=====] - 0s 25us/sample - loss: 2.7035 - acc: 0.1531 - val_loss: 2.6799 -
Epoch 61/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.7039 - acc: 0.1564 - val_loss: 2.6807 -
Epoch 62/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7005 - acc: 0.1585 - val_loss: 2.6782 -
Epoch 63/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7013 - acc: 0.1566 - val_loss: 2.6791 -
Epoch 64/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7002 - acc: 0.1545 - val_loss: 2.6785 -
Epoch 65/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.7025 - acc: 0.1562 - val_loss: 2.6786
```

```
4204/4204 [-----] - 0s 22us/sample - loss: 2.7025 - acc: 0.1562 - val_loss: 2.6766 -  
Epoch 66/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6997 - acc: 0.1557 - val_loss: 2.6770 -  
Epoch 67/100  
4284/4284 [=====] - 0s 21us/sample - loss: 2.6965 - acc: 0.1559 - val_loss: 2.6785 -  
Epoch 68/100  
4284/4284 [=====] - 0s 23us/sample - loss: 2.6958 - acc: 0.1569 - val_loss: 2.6789 -  
Epoch 69/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6979 - acc: 0.1571 - val_loss: 2.6772 -  
Epoch 70/100  
4284/4284 [=====] - 0s 21us/sample - loss: 2.6985 - acc: 0.1548 - val_loss: 2.6785 -  
Epoch 71/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6956 - acc: 0.1559 - val_loss: 2.6820 -  
Epoch 72/100  
4284/4284 [=====] - 0s 27us/sample - loss: 2.6969 - acc: 0.1583 - val_loss: 2.6780 -  
Epoch 73/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6949 - acc: 0.1604 - val_loss: 2.6775 -  
Epoch 74/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6994 - acc: 0.1597 - val_loss: 2.6773 -  
Epoch 75/100  
4284/4284 [=====] - 0s 24us/sample - loss: 2.6928 - acc: 0.1559 - val_loss: 2.6787 -  
Epoch 76/100  
4284/4284 [=====] - 0s 23us/sample - loss: 2.6942 - acc: 0.1562 - val_loss: 2.6776 -  
Epoch 77/100  
4284/4284 [=====] - 0s 23us/sample - loss: 2.6944 - acc: 0.1578 - val_loss: 2.6762 -  
Epoch 78/100  
4284/4284 [=====] - 0s 25us/sample - loss: 2.6973 - acc: 0.1611 - val_loss: 2.6786 -  
Epoch 79/100  
4284/4284 [=====] - 0s 24us/sample - loss: 2.6963 - acc: 0.1573 - val_loss: 2.6767 -  
Epoch 80/100  
4284/4284 [=====] - 0s 24us/sample - loss: 2.6892 - acc: 0.1587 - val_loss: 2.6752 -  
Epoch 81/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6945 - acc: 0.1611 - val_loss: 2.6767 -  
Epoch 82/100  
4284/4284 [=====] - 0s 23us/sample - loss: 2.6936 - acc: 0.1608 - val_loss: 2.6769 -  
Epoch 83/100  
4284/4284 [=====] - 0s 22us/sample - loss: 2.6965 - acc: 0.1580 - val_loss: 2.6782 -  
Epoch 84/100  
4284/4284 [=====] - 0s 21us/sample - loss: 2.6873 - acc: 0.1627 - val_loss: 2.6777 -  
Epoch 85/100  
4284/4284 [=====] - 0s 23us/sample - loss: 2.6900 - acc: 0.1524 - val_loss: 2.6766 -  
Epoch 86/100  
4284/4284 [=====] - 0s 24us/sample - loss: 2.6894 - acc: 0.1578 - val_loss: 2.6783 -  
Epoch 87/100
```



```

4284/4284 [=====] - 0s 23us/sample - loss: 2.6881 - acc: 0.1569 - val_loss: 2.6801 -
Epoch 88/100
4284/4284 [=====] - 0s 24us/sample - loss: 2.6917 - acc: 0.1550 - val_loss: 2.6803 -
Epoch 89/100
4284/4284 [=====] - 0s 25us/sample - loss: 2.6925 - acc: 0.1573 - val_loss: 2.6774 -
Epoch 90/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.6882 - acc: 0.1559 - val_loss: 2.6777 -
Epoch 91/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.6897 - acc: 0.1564 - val_loss: 2.6801 -
Epoch 92/100
4284/4284 [=====] - 0s 26us/sample - loss: 2.6876 - acc: 0.1639 - val_loss: 2.6779 -
Epoch 93/100
4284/4284 [=====] - 0s 25us/sample - loss: 2.6831 - acc: 0.1545 - val_loss: 2.6785 -
Epoch 94/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.6886 - acc: 0.1592 - val_loss: 2.6786 -
Epoch 95/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.6839 - acc: 0.1657 - val_loss: 2.6800 -
Epoch 96/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.6866 - acc: 0.1597 - val_loss: 2.6797 -
Epoch 97/100
4284/4284 [=====] - 0s 27us/sample - loss: 2.6877 - acc: 0.1557 - val_loss: 2.6794 -
Epoch 98/100
4284/4284 [=====] - 0s 22us/sample - loss: 2.6842 - acc: 0.1632 - val_loss: 2.6778 -
Epoch 99/100
4284/4284 [=====] - 0s 21us/sample - loss: 2.6837 - acc: 0.1576 - val_loss: 2.6782 -
Epoch 100/100
4284/4284 [=====] - 0s 23us/sample - loss: 2.6860 - acc: 0.1578 - val_loss: 2.6790 -

```

```

1 test = np.array( [[1 , 82.043 ,61.014 ,60.486 ,0 ,0 ,0 ,0, 0, 0]] )
2
3 result = model.predict(test)
4 print(result)

```

```
↳ [[2.6206630e-25 1.0000000e+00 1.5599000e-31 0.0000000e+00 0.0000000e+00  
0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00  
0.0000000e+00 0.0000000e+00 2.0089665e-38 3.3201757e-27 4.3253647e-23  
1.2997942e-25 3.5848725e-36 9.2289560e-33 4.0921082e-25 1.2057035e-19  
2.4889662e-26 7.4777198e-24 3.2222771e-18 3.9813497e-18 3.4221939e-15]]
```