**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/@timothychong/talk-data-to-me-modeling-of-singapore-grand-prix-201 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 Acqusition, Logistic Regression, SVM**

Data Acqusition 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 190( 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 ( Prix wins.

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

2014 - Current: V6 hybrid engines

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

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g

Enter your authorization code:
··········
Mounted at /content/drive
```

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

```
import requests
import json
import copy


def convertTimeToInt(time):
  time = [char for char in time]
  time[4] = ":"
  time = "".join(time)
  (m, s, ms) = time.split(':')
  result = float(m) * 60 + float(s) + float(s) * 1/100

  return (str(result))

#get race results of a specific season
def getRaceResultErgast(season,CSV):


  raceResults = requests.get('http://ergast.com/api/f1/{}/results.json?limit=1000'.format(season))
  qualiResults = requests.get('http://ergast.com/api/f1/{}/qualifying.json?limit=1000'.format(season))

  qualiResults = qualiResults.json()
  results = raceResults.json()
  raceCount = 0
  championshipPoints = {}
  oldConstructorPoints = {}
  constructorPoints = {}
  championshipVic = {}
  championshipPole = {}
  constructorVic = {}
  constructorPole = {}


  for i in results['MRData']['RaceTable']['Races']:
    raceDict = {}
    champWin = 0
    qualiWin = 0

```

```
    #for quali results for each race
    for j in qualiResults['MRData']['RaceTable']['Races'][raceCount]['QualifyingResults']:
      name = j['Driver']['givenName'] + ' ' + j['Driver']['familyName']

      if qualiWin == 0:
        if name not in championshipPole:
          championshipPole[name] = 1
        else:
          championshipPole[name] += 1

        if j['Constructor']['name'] not in constructorPole:
          constructorPole[ j['Constructor']['name'] ] = 1
        else:
          constructorPole[ j['Constructor']['name'] ] += 1

      try:
        q1 = ( j['Q1'] )
        q1 = convertTimeToInt(q1)
      except:
        q1 = '0:00:000'
        q1 = convertTimeToInt(q1)
      try:
        q2 = ( j['Q2'] )
        q2 = convertTimeToInt(q2)
      except:
        q2 = q1
      try:
        q3 = ( j['Q3'] )
        q3 = convertTimeToInt(q3)
      except:
        q3 = q2
      quali = [q1,q2,q3]
      raceDict[name] = quali
      qualiWin += 1

      #get constructor standing before each race

    for j in i['Results']:
      # print(j)
```

```
    # print(j['position'] + j['grid'] + j['Driver']['givenName'] + j['Driver']['familyName'] + j['Constructor'
    name = j['Driver']['givenName'] + ' ' + j['Driver']['familyName']
    constructorName = j['Constructor']['name']

    if champWin == 0:
      if name not in championshipVic:
        championshipVic[name] = 1
      else:
        championshipVic[name] += 1

      if constructorName not in constructorVic:
        constructorVic[ constructorName ] = 1
      else:
        constructorVic[ constructorName ] += 1

    row = []
    row.append(str(season))
    row.append(i['raceName'])
    row.append(j['position'])
    row.append(j['grid'])
    try:
      quali = raceDict[name]
      row.append(quali[0])
      row.append(quali[1])
      row.append(quali[2])
    except:
      row.append('0')
      row.append('0')
      row.append('0')
    row.append(name)
    row.append(j['Constructor']['name'])


     # no points for first race
    if raceCount == 0:
      row.append('0')
      row.append('0')
      row.append('0')
      row.append('0')
      row.append('0')
```

```
        row.append('0')
        row.append('0')
      else:
        try:
          row.append(str(championshipPoints[name]))
        except:
          row.append('0')
        try:
          row.append(str(championshipVic[name]))
        except:
          row.append('0')
        try:
          row.append(str(championshipPole[name]))
        except:
          row.append('0')
        try:
          row.append(str(oldConstructorPoints[constructorName]))
        except:
          row.append('0')
        try:
          row.append(str(constructorVic[constructorName]))
        except:
          row.append('0')
        try:
          row.append(str(constructorPole[constructorName]))
        except:
          row.append('0')

      if name not in championshipPoints:
        championshipPoints[name] = float(j['points'])
      else:
        championshipPoints[name] += float(j['points'])

      if constructorName not in constructorPoints:
        constructorPoints[constructorName] = float(j['points'])
      else:
        constructorPoints[constructorName] += float(j['points'])


      row = ','.join(row)
```

```
      row = ','.join(row)
      print(row, file = CSV)
      champWin +=1
    raceCount+=1
    oldConstructorPoints = copy.deepcopy(constructorPoints)

#get desired results and parse it into CSV
def getResults():
  resultCSV = open('drive/My Drive/results.csv', 'w')
  print("Year,Race,Final Position,Qualifying Position,Q1,Q2,Q3,Name,Team, Championship Points, Race Vic, Race P


  for i in range(2006,2019):
    getRaceResultErgast(str(i),resultCSV)

  resultCSV.close()

getResults()

```

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import linear_model, preprocessing
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support
import warnings
warnings.filterwarnings("ignore")
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import seaborn as sn
import pandas as pd
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-lElfrMvCqM8kyIVKsRr/view?usp=sharing](https://drive.google.com/file/d/1WwxQ8t22FjyL-lElfrMvCqM8kyIVKsRr/view?usp=sharing) for access

```
df = pd.read_excel("drive/My Drive/results.xlsx")
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 | 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

```
#generating labels for names/teams/races
races = np.unique(df['Race'])
# print(races)

LE = LabelEncoder()
race_labels = LE.fit_transform(df['Race'])
race_mappings = {index: label for index, label in enumerate(LE.classes_)}
# print(race_mappings)
df['RaceLabel'] = race_labels
```

```
df['RaceLabel'] = race_labels

names = np.unique(df['Name'])
name_labels = LE.fit_transform(df['Name'])
name_mappings = {index: label for index, label in enumerate(LE.classes_)}
# print(name_mappings)
df['NameLabel'] = name_labels

teams = np.unique(df['Team'])
team_labels = LE.fit_transform(df['Team'])
team_mappings = {index: label for index, label in enumerate(LE.classes_)}
# print(team_mappings)
df['TeamLabel'] = team_labels

df.head()
```

| | Year | Race | Final Position | Qualifying Position | Q1 | Q2 | Q3 | Name | Team | Championship Points | Race Vic | Race Pole |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **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 |
| **4** | 2006 | Bahrain Grand Prix | 5 | 5 | 93.33 | 91.31 | 92.32 | Juan Pablo Montoya | McLaren | 0.0 | 0 | 0 |

Target Feature = Final Position

Features = Everything else

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

## Initial Accuracy with just tran_test_split

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=0)

logreg = linear_model.LogisticRegression(C=1)
logreg.fit(x_train,y_train)

yhat = logreg.predict(x_test)
accuracy = np.mean(yhat == y_test)
print("Accuracy of training = {}".format(accuracy))
```

```
Accuracy of training = 0.12943372744243933
```

## Calculate mean accuracy and standardard error with KFold cross validation

```
nfold = 10
kf = KFold(n_splits=nfold, shuffle=True)
acc = []

for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Fit a model
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)

    # Measure performance
    acc.append(np.mean(yhat == yts))


```

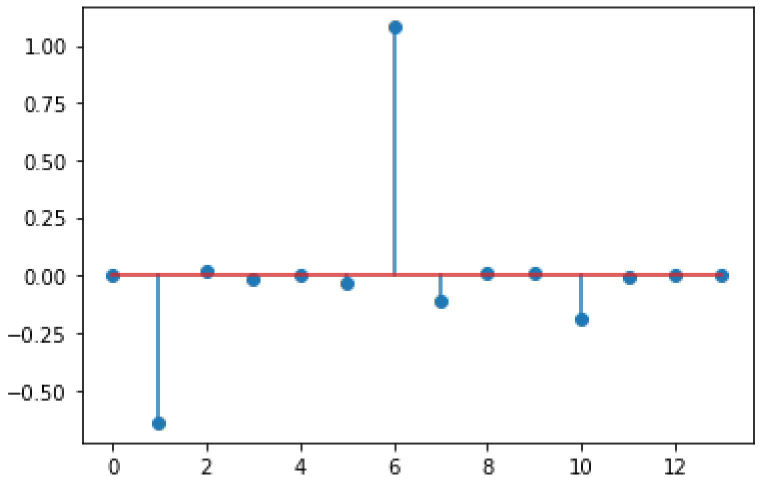
```
SE = np.std(acc) / np.sqrt(nfold)
print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
```

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

Weights of each feature from the logistic regression

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

```

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




Features = *Year*, Qualifying Position, Q1, Q2, Q3 ,Championship Points, Race Vic, Race Pole,Constructor Points, Team Vic, Team Pole , Ra , 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 domina the car and less by the skills of the driver.

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

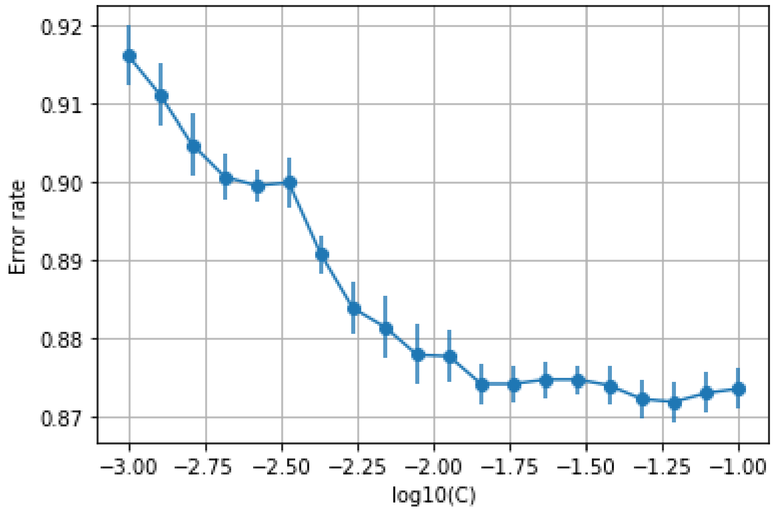
We're also optimizing L1 by finding an optimal C

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

# Create the cross-validation object and error rate matrix
nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
err_rate = np.zeros((npen,nfold))
num_nonzerocoef = np.zeros((npen,nfold))
# Create the logistic regression object
logreg = linear_model.LogisticRegression(penalty='l1',warm_start=True)

# Loop over the folds in the cross-validation
for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Loop over penalty levels
    for ipen, c in enumerate(C_test):

        # Set the penalty level
        logreg.C= c

        # Fit a model on the training data
        logreg.fit(Xtr, ytr)

        # Predict the labels on the test set.
        yhat = logreg.predict(Xts)

        # Measure the accuracy
        err_rate[ipen,ifold] = np.mean(yhat != yts)
```

```

    print("Fold %d" % ifold)
```

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

```
err_mean = np.mean(err_rate, axis=1)
err_se = np.std(err_rate,axis=1)/np.sqrt(nfold-1)
plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
plt.xlabel('log10(C)')
plt.ylabel('Error rate')
plt.grid()
plt.show()
imin = np.argmin(err_mean)

print("The minimum test error rate = {}, SE={}".format(err_mean[imin], err_se[imin]))
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

```
logreg = linear_model.LogisticRegression(C=C_test[imin],penalty='l1')
confusion = np.zeros((24,24))
nfold = 10
kf = KFold(n_splits=nfold, shuffle=True)
acc = []
# 8 classes so 8 for matrix size

for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Fit a model
    logreg.fit(Xtr, ytr)
```

```
    yhat = logreg.predict(Xts)

    # Measure performance
    acc.append(np.mean(yhat == yts))
    confusion += confusion_matrix(yts,yhat)


SE = np.std(acc) / np.sqrt(nfold)
tempSum = np.sum(confusion,1)
confusion = confusion / tempSum[np.newaxis,:]
print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))


```

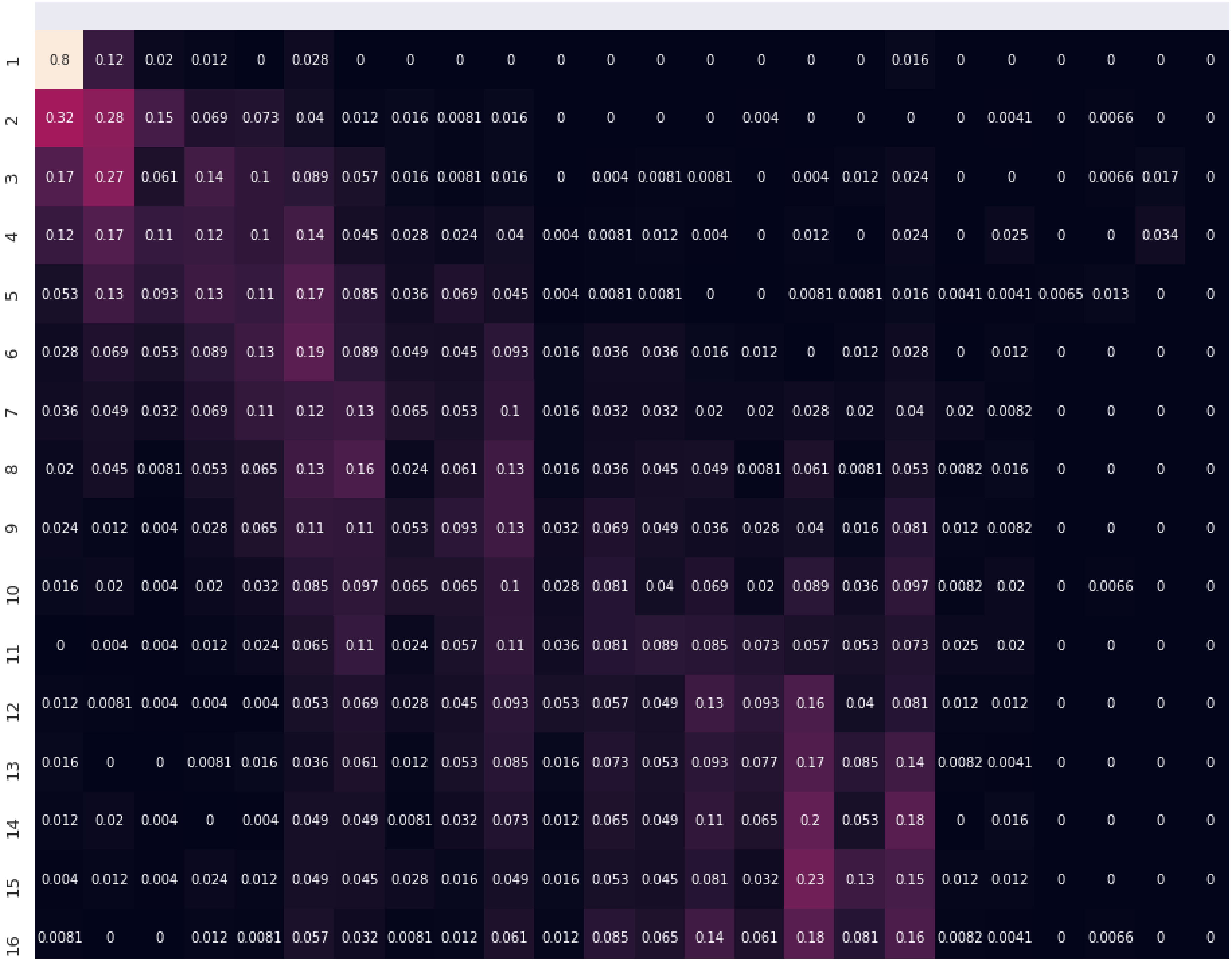
```
Mean Accuracy rate = 0.12624145627005162 , SE = 0.00509793928972676
```

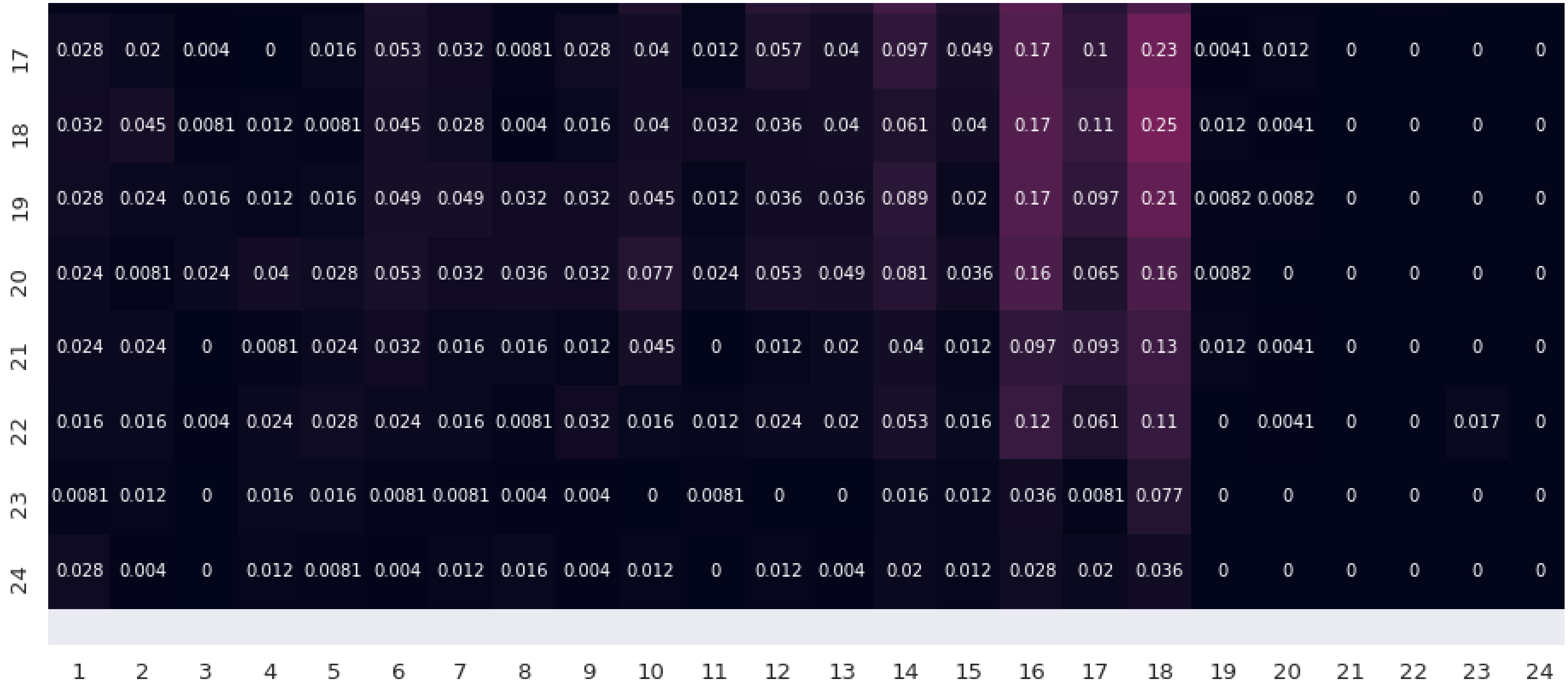
Our mean accuracy slightly improved with optimal L1 regularization from 0.12605454038220115 to 0.12624145627005162

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

```
df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,2
                  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]])
plt.figure(figsize = (20,20))

sn.set(font_scale=1.2)#for label size
ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
bottom, top = ax.get_ylim()
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 unpredictiability in the mid-field teams. Many things race can occur in a race so it is no surprise that the accuracy is quite low.

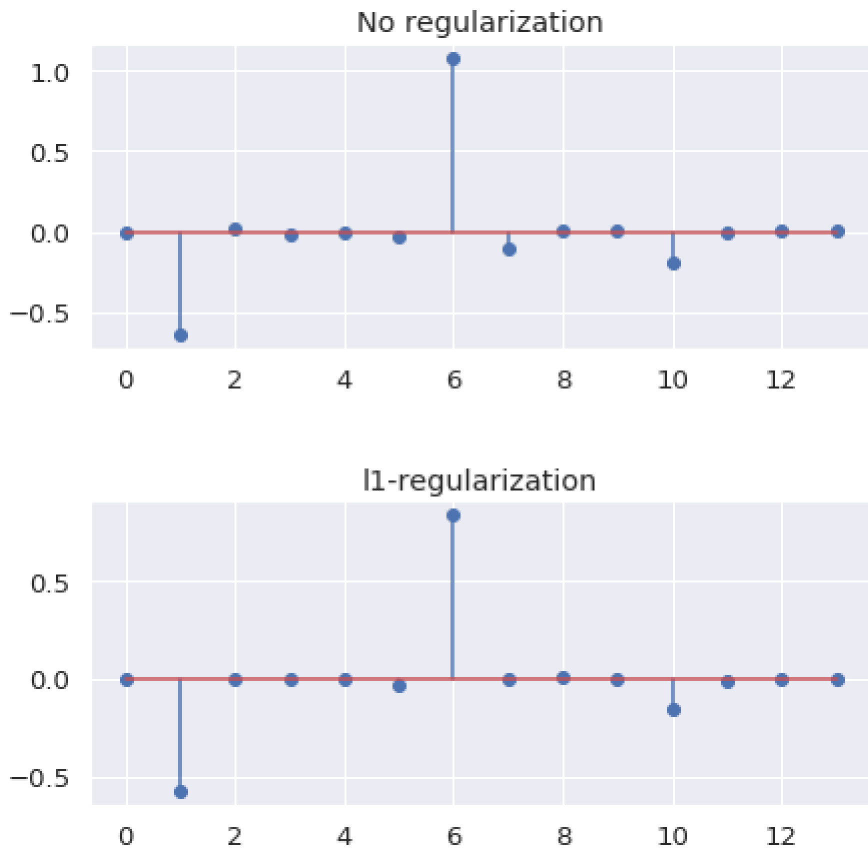
What's suprising about this is that it does pretty poorly predicting second place

Comparsion of weights without L1 and with L1 regularixation

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

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

```
Text(0.5, 1.0, 'l1-regularization')
```

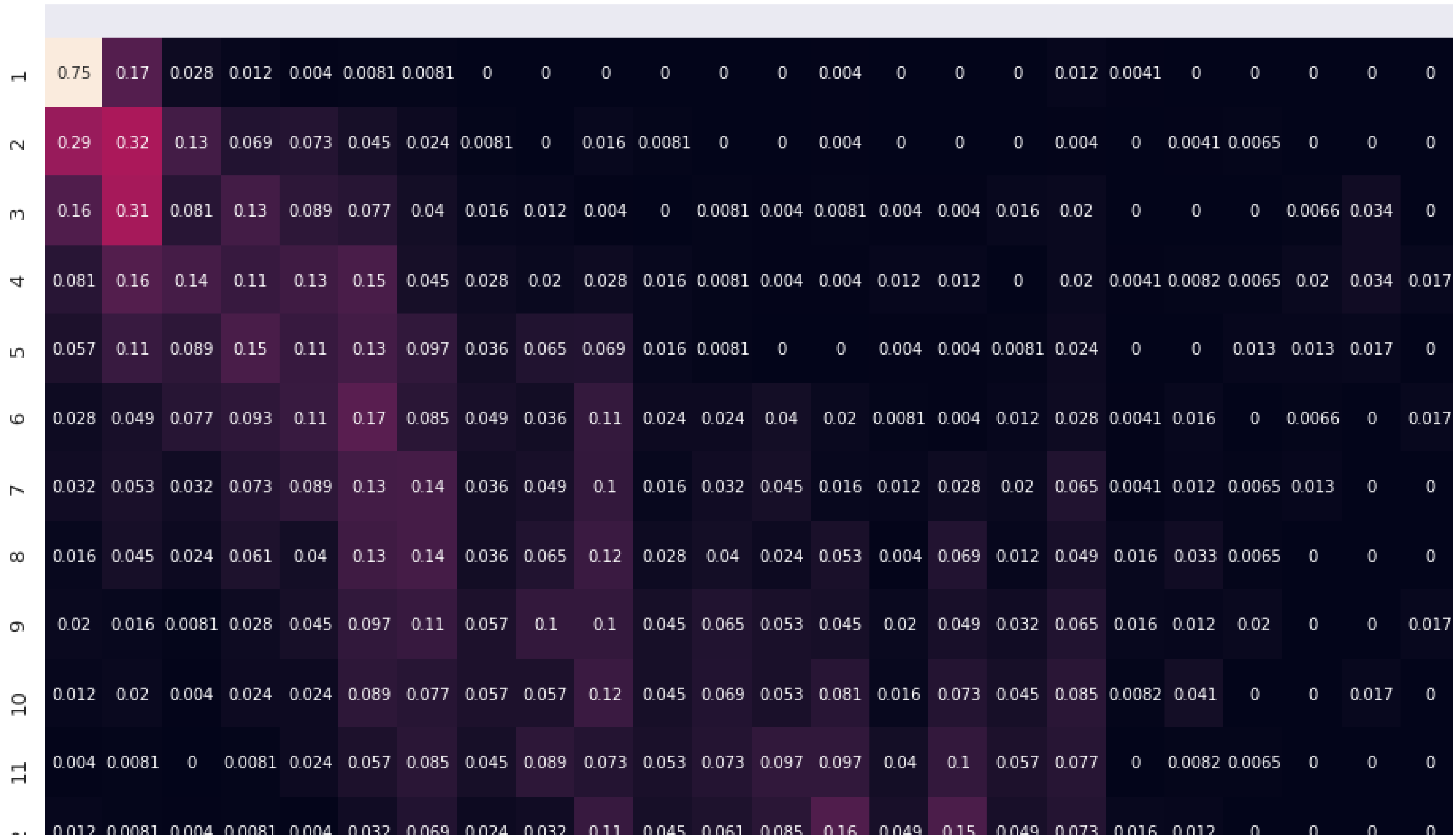



Now we will try a multi-class logistic regression

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

```

for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Fit a model
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)

    # Measure performance
    acc.append(np.mean(yhat == yts))
    confusion += confusion_matrix(yts,yhat)

    print("Fold %d" % ifold)


SE = np.std(acc) / np.sqrt(nfold)
tempSum = np.sum(confusion,1)
confusion = confusion / tempSum[np.newaxis,:]
print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))

df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,2
                  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]])
plt.figure(figsize = (20,20))

sn.set(font_scale=1.2)#for label size
ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
bottom, top = ax.get_ylim()
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)
```



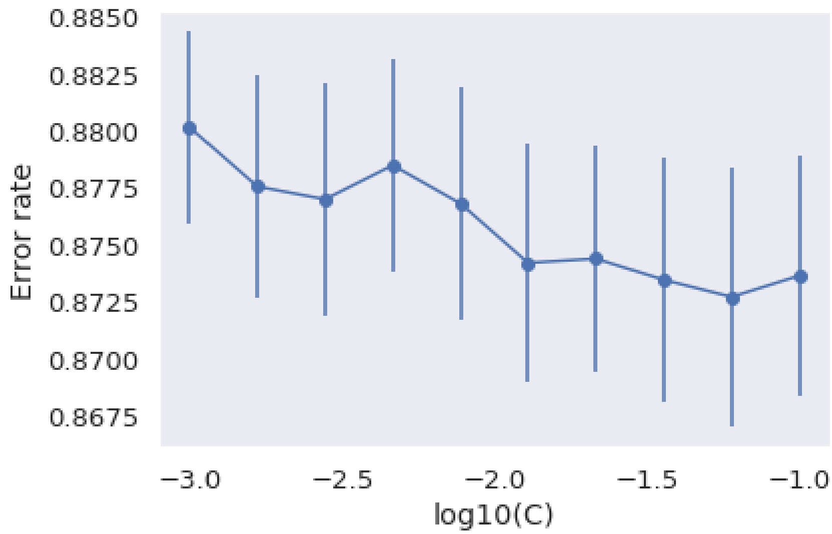

Mean accuracy using multi-class model performed slightly worse with 0.12492676802901378 compared to logistic regression of 0.12624145627005162

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

```
npen = 10
C_test = np.logspace(-3,-1,npen)
# Create the cross-validation object and error rate matrix
nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
err_rate = np.zeros((npen,nfold))
num_nonzerocoef = np.zeros((npen,nfold))
# Create the logistic regression object
logreg = linear_model.LogisticRegression(solver='newton-cg', multi_class='multinomial', penalty='l2')
# Loop over the folds in the cross-validation
for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Loop over penalty levels
    for ipen, c in enumerate(C_test):

        # Set the penalty level
        logreg.C= c

        # Fit a model on the training data
        logreg.fit(Xtr, ytr)

        # Predict the labels on the test set.
        yhat = logreg.predict(Xts)

        # Measure the accuracy
        err_rate[ipen,ifold] = np.mean(yhat != yts)
    print("Fold %d" % ifold)
```

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

```
err_mean = np.mean(err_rate, axis=1)
err_se = np.std(err_rate,axis=1)/np.sqrt(nfold-1)
plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
plt.xlabel('log10(C)')
plt.ylabel('Error rate')
plt.grid()
plt.show()
imin = np.argmin(err_mean)

print("The minimum test error rate = {}, SE={}".format(err_mean[imin], err_se[imin]))
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
```

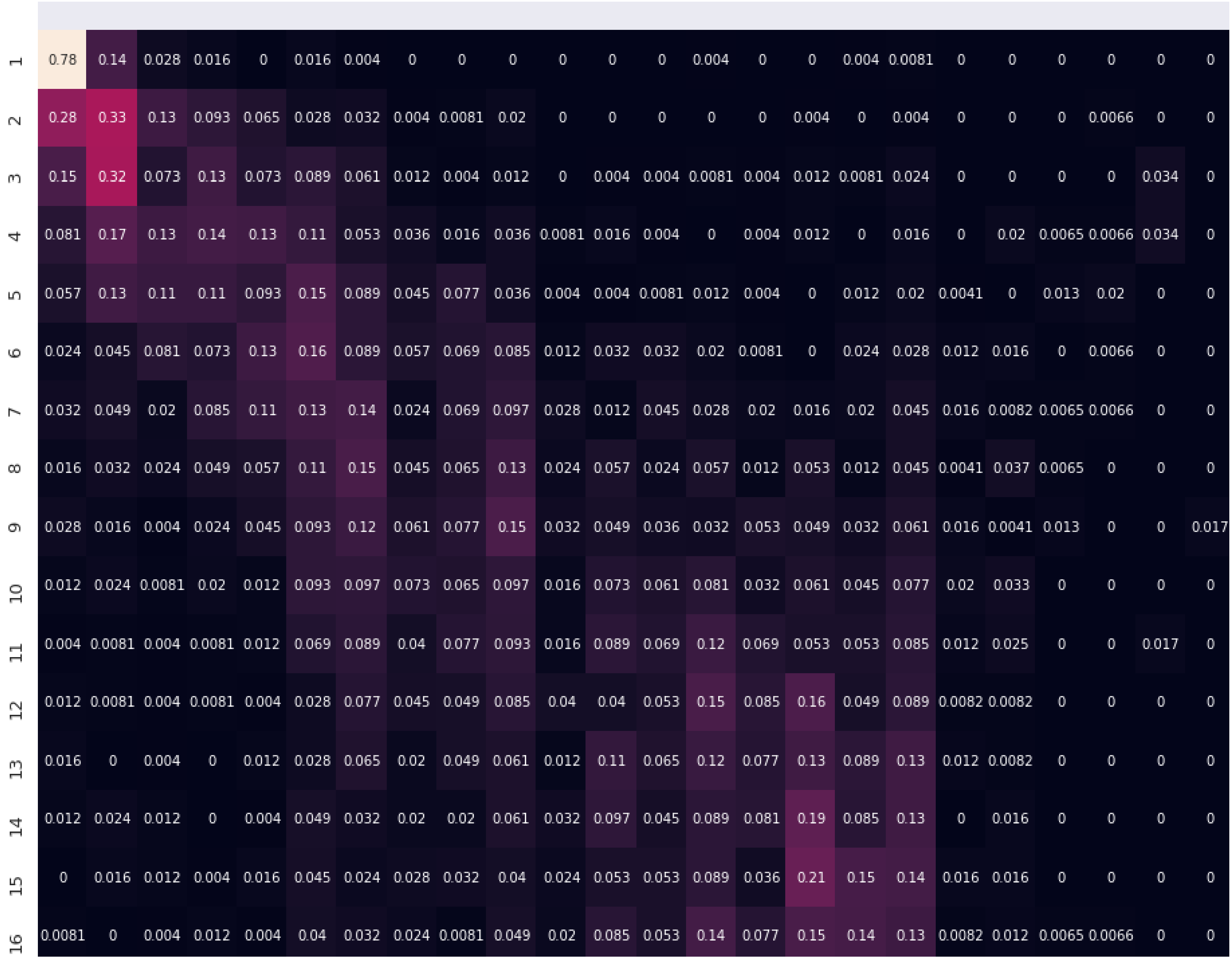
```
logreg = linear_model.LogisticRegression(C=C_test[imin], solver='newton-cg', multi_class='multinomial', penalty
confusion = np.zeros((24,24))
nfold = 10
kf = KFold(n_splits=nfold, shuffle=True)
acc = []
# 8 classes so 8 for matrix size

for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Fit a model
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)

```

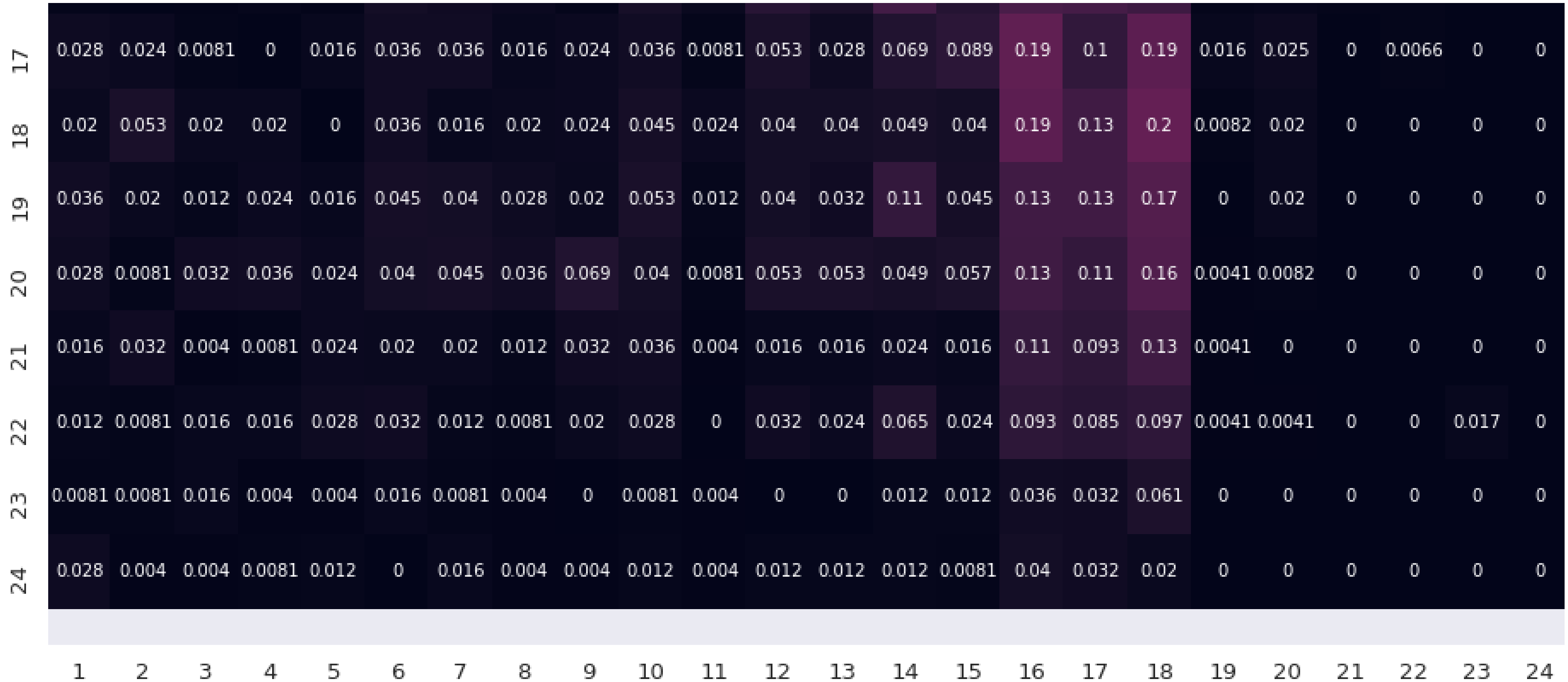
```
    # Measure performance
    acc.append(np.mean(yhat == yts))
    confusion += confusion_matrix(yts,yhat)


SE = np.std(acc) / np.sqrt(nfold)
tempSum = np.sum(confusion,1)
confusion = confusion / tempSum[np.newaxis,:]
print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
```

```
Mean Accuracy rate = 0.12174815176454179 , SE = 0.003204075834675082
```

```
df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,2
                  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]])
plt.figure(figsize = (20,20))

sn.set(font_scale=1.2)#for label size
ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
bottom, top = ax.get_ylim()
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

```
from sklearn import svm
svc = svm.SVC(probability=False,  kernel="rbf", C=2.8, gamma=.0073)

nfold = 10
kf = KFold(n_splits=nfold, shuffle=True)
acc = []
# 8 classes so 8 for matrix size

for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
```

```
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Fit a model
    svc.fit(Xtr, ytr)
    yhat = svc.predict(Xts)

    # Measure performance
    acc.append(np.mean(yhat == yts))

    print("Fold: %d" %ifold)


SE = np.std(acc) / np.sqrt(nfold)
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 becausse the slack from SVM reduces the accuracy of predicting the exact position be the difference betwen 5th-7th in the mid-field might be minimal.

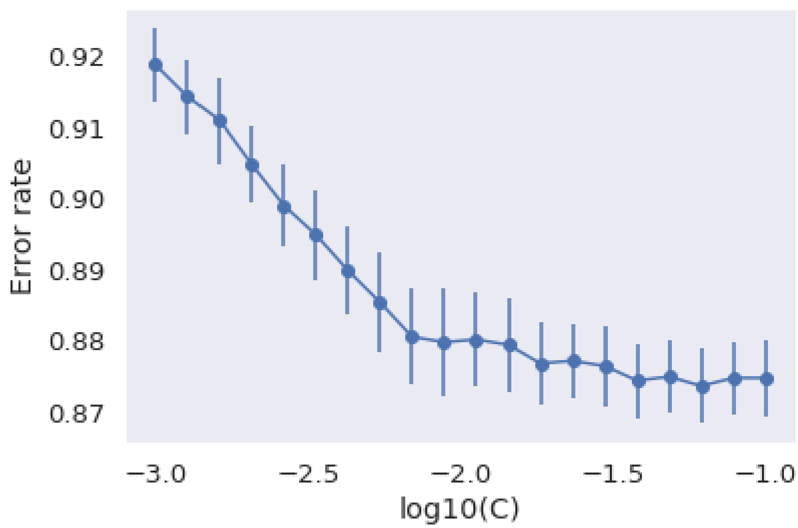
### Conclusion: Logistic Regession with L1 regularization seems to perform the best

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

```

```
# Create the cross-validation object and error rate matrix
nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
err_rate = np.zeros((npen,nfold))
num_nonzerocoef = np.zeros((npen,nfold))
# Create the logistic regression object
logreg = linear_model.LogisticRegression(penalty='l1',warm_start=True)

# Loop over the folds in the cross-validation
for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
    Xts = x[Its,:]
    yts = y[Its]

    # Loop over penalty levels
    for ipen, c in enumerate(C_test):

        # Set the penalty level
        logreg.C= c

        # Fit a model on the training data
        logreg.fit(Xtr, ytr)

        # Predict the labels on the test set.
        yhat = logreg.predict(Xts)

        # Measure the accuracy
        err_rate[ipen,ifold] = np.mean(yhat != yts)

    print("Fold %d" % ifold)

```

```
err_mean = np.mean(err_rate, axis=1)
err_se = np.std(err_rate,axis=1)/np.sqrt(nfold-1)
plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
```

```
plt.xlabel('log10(C)')
plt.ylabel('Error rate')
plt.grid()
plt.show()
imin = np.argmin(err_mean)

print("The minimum test error rate = {}, SE={}".format(err_mean[imin], err_se[imin]))
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
```

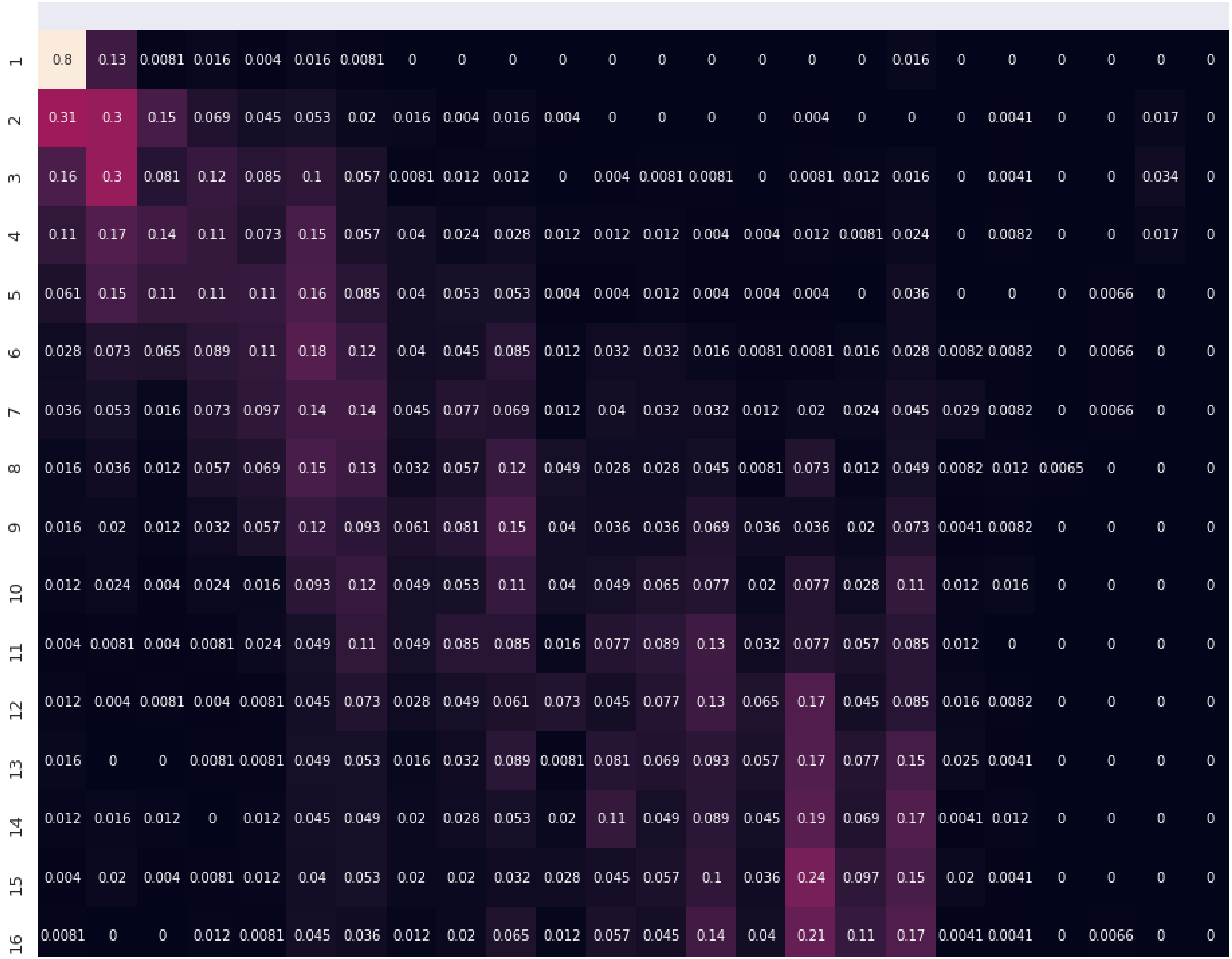
```
logreg = linear_model.LogisticRegression(C=C_test[imin],penalty='l1')
confusion = np.zeros((24,24))
nfold = 10
kf = KFold(n_splits=nfold, shuffle=True)
acc = []

for ifold, Ind in enumerate(kf.split(x)):

    # Get training and test data
    Itr, Its = Ind
    Xtr = x[Itr,:]
    ytr = y[Itr]
```

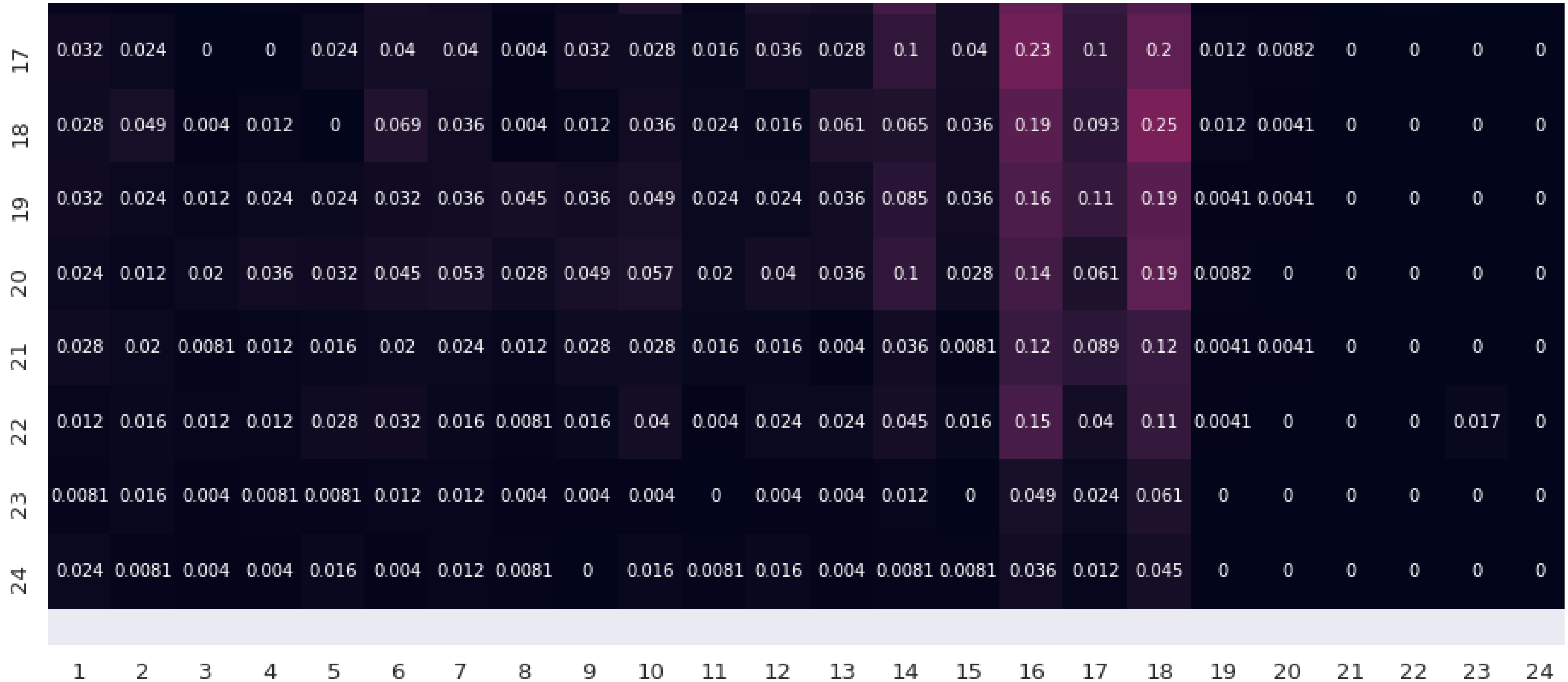
```
    Xts = x[Its,:]
    yts = y[Its]

    # Fit a model
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)

    # Measure performance
    acc.append(np.mean(yhat == yts))
    confusion += confusion_matrix(yts,yhat)


SE = np.std(acc) / np.sqrt(nfold)
tempSum = np.sum(confusion,1)
confusion = confusion / tempSum[np.newaxis,:]
print("Mean Accuracy rate = {} , SE = {}".format(np.mean(acc),SE))
```

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

```
df_cm = pd.DataFrame(confusion, index = [i for i in [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,2
                  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]])
plt.figure(figsize = (20,20))

sn.set(font_scale=1.2)#for label size
ax = sn.heatmap(df_cm, annot=True,annot_kws={"size": 10})
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
```

```
(24.5, -0.5)
```


1 0.8 0.13 0.0081 0.016 0.004 0.016 0.0081 0 0 0 0 0 0 0 0 0 0 0.016 0 0 0 0 0 0
2 0.31 0.3 0.15 0.069 0.045 0.053 0.02 0.016 0.004 0.016 0.004 0 0 0 0 0.004 0 0 0 0.0041 0 0 0.017 0
3 0.16 0.3 0.081 0.12 0.085 0.1 0.057 0.0081 0.012 0.012 0 0.004 0.0081 0.0081 0 0.0081 0.012 0.016 0 0.0041 0 0 0.034 0
4 0.11 0.17 0.14 0.11 0.073 0.15 0.057 0.04 0.024 0.028 0.012 0.012 0.012 0.004 0.004 0.012 0.0081 0.024 0 0.0082 0 0 0.017 0
5 0.061 0.15 0.11 0.11 0.11 0.16 0.085 0.04 0.053 0.053 0.004 0.004 0.012 0.004 0.004 0.004 0 0.036 0 0 0 0.0066 0 0
6 0.028 0.073 0.065 0.089 0.11 0.18 0.12 0.04 0.045 0.085 0.012 0.032 0.032 0.016 0.0081 0.0081 0.016 0.028 0.0082 0.0082 0 0.0066 0 0
7 0.036 0.053 0.016 0.073 0.097 0.14 0.14 0.045 0.077 0.069 0.012 0.04 0.032 0.032 0.012 0.02 0.024 0.045 0.029 0.0082 0 0.0066 0 0
8 0.016 0.036 0.012 0.057 0.069 0.15 0.13 0.032 0.057 0.12 0.049 0.028 0.028 0.045 0.0081 0.073 0.012 0.049 0.0082 0.012 0.0065 0 0 0
9 0.016 0.02 0.012 0.032 0.057 0.12 0.093 0.061 0.081 0.15 0.04 0.036 0.036 0.069 0.036 0.036 0.02 0.073 0.0041 0.0082 0 0 0 0
10 0.012 0.024 0.004 0.024 0.016 0.093 0.12 0.049 0.053 0.11 0.04 0.049 0.065 0.077 0.02 0.077 0.028 0.11 0.012 0.016 0 0 0 0
11 0.004 0.0081 0.004 0.0081 0.024 0.049 0.11 0.049 0.085 0.085 0.016 0.077 0.089 0.13 0.032 0.077 0.057 0.085 0.012 0 0 0 0 0
12 0.012 0.004 0.0081 0.004 0.0081 0.045 0.073 0.028 0.049 0.061 0.073 0.045 0.077 0.13 0.065 0.17 0.045 0.085 0.016 0.0082 0 0 0 0
13 0.016 0 0 0.0081 0.0081 0.049 0.053 0.016 0.032 0.089 0.0081 0.081 0.069 0.093 0.057 0.17 0.077 0.15 0.025 0.0041 0 0 0 0
14 0.012 0.016 0.012 0 0.012 0.045 0.049 0.02 0.028 0.053 0.02 0.11 0.049 0.089 0.045 0.19 0.069 0.17 0.0041 0.012 0 0 0 0
15 0.004 0.02 0.004 0.0081 0.012 0.04 0.053 0.02 0.02 0.032 0.028 0.045 0.057 0.1 0.036 0.24 0.097 0.15 0.02 0.0041 0 0 0 0
16 0.0081 0 0 0.012 0.0081 0.045 0.036 0.012 0.02 0.065 0.012 0.057 0.045 0.14 0.04 0.21 0.11 0.17 0.0041 0.0041 0 0.0066 0 0




```
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 , Ra
, 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

```
test = [[2019 , 1 , 82.043 ,61.014 ,60.486 ,0 ,0 ,0 ,0, 0, 0 , 1, 40, 14 ]]
result = logreg.predict(test)
print(result+1)

proba = logreg.predict_proba(test)
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

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

proba = logreg.predict_proba(test)
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**

```
import tensorflow as tf
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
import xlrd

from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras import optimizers
import tensorflow.keras.backend as K
```

```
data = pd.read_excel("drive/My Drive/results.xlsx")

df = pd.DataFrame(data, columns=["Year","Race","Final Position",
                                 "Qualifying Position","Q1","Q2","Q3",
                                 "Name","Team","Championship Points",
                                 "Race Vic","Race Pole","Constructor Points",
                                 "Team Vic","Team Pole"])
#print(df.isnull().any())

input_data = df[["Qualifying Position","Q1","Q2","Q3",
                "Championship Points","Race Vic",
                "Race Pole","Constructor Points",
                "Team Vic","Team Pole","Final Position"]]

n = input_data.shape[0]
input_data = input_data.values

tr_start = 0
tr_end = int(np.floor(0.8*n))
ts_start = tr_end + 1
ts_end = n

tr_data = input_data[tr_start:tr_end]
ts_data = input_data[ts_start:ts_end]

xtr = tr_data[:,0:-1]
ytr = tr_data[:,-1]
```

```
xts = ts_data[:,0:-1]
yts = ts_data[:,-1]

xmean = np.mean(xtr,axis=0)
xstd = np.std(xtr,axis=0)
#print(xmean, xstd)
xtr_scale = (xtr-xmean[None,:])/xstd[None,:]
xts_scale = (xts-xmean[None,:])/xstd[None,:]

K.clear_session()

nin = xtr.shape[1]
nout = np.max(ytr)+1
nh = round((nin+nout)/2)
#nh = 256
print(nh, nin, nout)
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation="relu", name="hidden"))
model.add(BatchNormalization())
model.add(Dense(units=nout, activation="softmax", name="output"))
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
_________________________________________________________________
```

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

```
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])

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 22us/sample - loss: 2.7025 - acc: 0.1562 - val_loss: 2.6786
```

```
4284/4284 [==============================] - 0s 22us/sample - loss: 2.7025 - acc: 0.1562 - val_loss: 2.6786 -
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 -
```

```
test = np.array( [[1 , 82.043 ,61.014 ,60.486 ,0 ,0 ,0 ,0, 0, 0]] )

result = model.predict(test)
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]]
```