Andrew Qu and Rihui Zheng

Professor Linda Sellie

CS-UY 4563 Introduction to Machine Learning

Final Project Write-up

INTRODUCTION

The dataset we are using is an NBA stats dataset consisting of all the player stats across all seasons of the NBA for the past few decades. The data can be found here.

Problem Statement

In professional basketball, there are five unique positions a player can play as. Those positions are center, power forward, point guard, shooting guard, and small forward. Anyone who does not watch a lot of basketball may not know the difference among these positions. The problem we are trying to solve is how different these positions are with each other. What kind of quality and skills does a player need to excel at a position? Are there even unique traits that certain positions have over others? Given a player's statistics, how accurately can a machine learning model predict their position? If our model is good, it even could tell us what role a certain player should play given their skill stats.

Preprocessing

The initial dataset has roughly 25 thousand rows and over 50 columns. Each row is defined by a year, a player, and their team. For example, there would be one row to represent Kobe Bryant's 2010 season with the LA Lakers. It's possible for a player and year to appear in more than row, given that they were traded to a different team that year.

Since the dataset was large, we wanted to take only a sample of it to save on runtime. Firstly, we dropped any rows whose 'Pos' or 'Position' was not one of the main five: PG, SG, SF, PF, or C. This dropped less than 100 rows, so we still needed to pare down. However, instead of using a random sample, we felt that we could keep the highest quality data by using the MP or 'minutes played' feature. A low MP signals that other features can be potentially misleading. For example, imagine a bench player that only has ten minutes played over the season. In that short time, perhaps they attempted three field goals and made two, resulting in a FG% of 0.66. That's an excellent FG%, but on a very small sample size. Therefore, we dropped rows using a threshold on 'MP' which we tuned until we had a desirable number of rows left.

Many features were dropped for different reasons. The player's name of course has no impact on their gameplay, and while their team could potentially affect their role or playstyle, there are too many to encode. We also dropped features that were stat totals, like PTS, AST, and STL (Points, Assists, and Steals). These are essential stats, but they depend on both position and playtime, and while we already processed on MP, some samples still had twice as many minutes as others. Therefore, we preferred efficiency, keeping features like AST%', which represents the percentage of field goals that the player assisted on *only while he was on the court*. Other features were eliminated because they were too complex and/or only measure how "good" a player is. Some examples of these are PER, WS, and VORP (Player Efficiency Rating, Win Shares, Value Over Replacement Player). We assumed good players would have similar values for these regardless of their positions, and it would also be hard for someone new to basketball to understand. The features that we decided to use were:

Pos - Position

Age - Player's age on Feb. 1st of that season

TS% - True shooting percentage

3PAr - 3 Point attempt rate, the percentage of field goal attempts that were for 3 points

TRB% - Total rebound percentage, the percentage of available rebounds that the player grabbed while on the court

AST% - Assist percentage, the percentage of team's fields goals that the player assisted on while on the court

STL% - Steal percentage, the percentage of enemy possessions that ended with a steal by the player while on the court

BLK% - Block percentage, the percentage of enemy 2-point field goal attempts that were blocked by the player while on the court

USG% - Usage percentage, an estimate of the percentage of team plays that involved the player while on the court

FG% - Field goal percentage

3P% - 3 point field goal percentage

2P% - 2 point field percentage

FT% - Free throw percentage

After dropping rows based on MP, dropping unused columns, and then dropping rows with missing data, we were left with 5101 samples. The positions were encoded from [C, PF, PG, SG, SF] to [0, 1, 2, 3, 4]. The true positions were distributed thusly:

C: 722

PF: 1010

PG: 1106

SG: 1133

SF: 1130

The feature matrix was scaled using sklearn.preprocessing.StandardScaler() with default parameters. The samples were split into training and testing data using sklearn.model_selection.train_test_split(), using test_size=0.33.

SUPERVISED ANALYSIS

For the project, we used three different learning models: Logistic Regression, Support Vector Machine, and Neural Networks. The code for all of the models and tests can be found here.

Logistic Regression

The first machine learning model we tried was logistic regression. We used the scikit-learn's LogisticRegression class to help create, train, and test the models.

We began with just a normal logistic regression model with no feature transformation and no regularization. The code we wrote to find the accuracy of the model was:

```
model = LogisticRegression(penalty='none', multi_class='multinomial', class_weight='balanced', max_iter=12000)
model.fit(X_train,y_train)
acc_test = model.score(X_test, y_test)
acc_train = model.score(X_train, y_train)
print("Test Accuracy:", acc_test, "\tTraining Accuracy:", acc_train)
```

This gave us a test accuracy of 70.84% and a training accuracy of 71.14%. We also took a look at the weights of this model for every feature of each position:

```
==============================
C
Age :  0.7080926459883256
TS% :  -0.07220445673065422
3PAr :  -0.42001261122246086
TRB% :  4.107131532345062
AST% :  -1.1707767302949321
STL% :  -1.2910532667088142
BLK% :  1.5744894320982539
USG% :  -0.006402095810550364
FG% :  0.9643271927194429
3P% :  0.04823838739939208
2P% :  -0.738198526868655
FT% :  -0.03714652497943109
==============================
PF
Age :  0.41083008659489084
TS% :  -0.17777024081623852
3PAr :  0.10150001471482296
TRB% :  3.51956052928679
AST% :  -1.4445829674008548
STL% :  -0.685604511507728
BLK% :  0.7685294252274804
USG% :  0.30498423330011487
FG% :  0.1876638806044154
3P% :  0.13110235411787566
2P% :  0.06018639788998158
FT% :  -0.09326751932084225
==============================
```

```
==============================
PG
Age :  -0.8301151995969149
TS% :  -0.037685828047466084
3PAr :  0.8196717917065519
TRB% :  -4.892110821188412
AST% :  3.48506431405382
STL% :  1.139238968267059
BLK% :  -2.17844248064951
USG% :  -0.8577922329227735
FG% :  0.17606383834027323
3P% :  -0.6375335428873408
2P% :  -0.40842073930903305
FT% :  0.1735472732554397
==============================
SG
Age :  -0.3197524030083236
TS% :  0.06421203303705775
3PAr :  -0.12973512670285184
TRB% :  -3.131460700258975
AST% :  0.039980665637708454
STL% :  0.7623976980837321
BLK% :  -0.4594597998224382
USG% :  0.29559995865041716
FG% :  -0.6382921877555163
3P% :  0.16828198727673227
2P% :  0.4803550243963564
FT% :  0.0371564177404416
==============================
```

```
==============================
SF
Age :  0.03094487002202223
TS% :  0.22344849255730861
3PAr :  -0.37142406849606296
TRB% :  0.3968794598155726
AST% :  -0.9096852819958001
STL% :  0.07502111186574192
BLK% :  0.2948834231462468
USG% :  0.26361013678281353
FG% :  -0.6897627239086022
3P% :  0.28991081409333996
2P% :  0.6060778438913633
FT% :  -0.08028964669559313
```

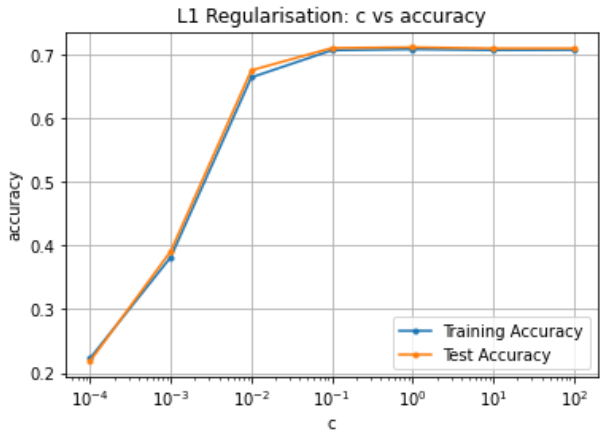
The weights tell us how important each feature is when deciding whether the player plays in a certain or not. A more in-depth discussion of what these weights tell us can be found in the conclusion section.

The training accuracy and testing accuracy were similar, so there doesn't seem to be too much overfitting. However, we still tried adding in regularization. We used the same scikit-learn class to try out both Lasso and Ridge regression with the logistic regression, and found the accuracies of these with various C values. The C values we tried were [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]. We used the model to find the training and test accuracies for each C value and plotted the accuracy vs. c values. The code we wrote to do all of this is:

```
# Try different C values if using penalty
def model_and_graph(penalty, solver, random_state, X_tr, y_tr, X_ts, y_ts, title):
    tr_acc = []
    ts_acc = []
    for c in cVals:
        model = LogisticRegression(penalty=penalty, C=c, solver=solver, random_state=random_state, \
                                   multi_class='multinomial', max_iter=12000)
        model.fit(X_tr,y_tr)
        ts_acc.append(model.score(X_ts, y_ts))
        tr_acc.append(model.score(X_tr, y_tr))
    for i in range(len(cVals)):
        print("C: ", cVals[i], " Train acc: ", tr_acc[i], " Test acc: ", ts_acc[i])
    plt.plot(cVals,tr_acc,'.-')
    plt.plot(cVals,ts_acc,'.-')
    plt.xlabel('c')
    plt.ylabel('accuracy')
    plt.xscale('log')
    plt.title(title)
    plt.grid()
    plt.legend(['Training Accuracy', 'Test Accuracy'])
    plt.show(block=False)
```

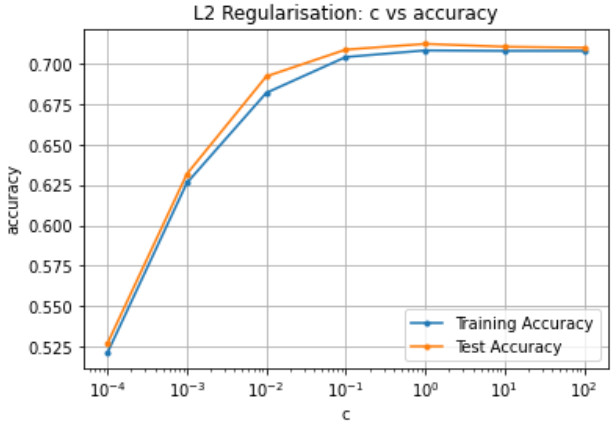
With lasso regression, the result we got was (next page):

```
C:  0.0001  Train acc:  0.22300263388937663  Test acc:  0.21852731591448932
C:  0.001  Train acc:  0.38191395961369623  Test acc:  0.3913301662707839
C:  0.01  Train acc:  0.6646180860403863  Test acc:  0.6757719714964371
C:  0.1  Train acc:  0.7073456248170911  Test acc:  0.7108076009501187
C:  1  Train acc:  0.7088088966930055  Test acc:  0.7119952494061758
C:  10  Train acc:  0.707638279192274  Test acc:  0.7102137767220903
C:  100  Train acc:  0.7079309335674568  Test acc:  0.7102137767220903
```



The best test accuracy was with a c value of 1, with a training accuracy of 70.88% and a test accuracy of 71.2%. With ridge regression the result we got was:

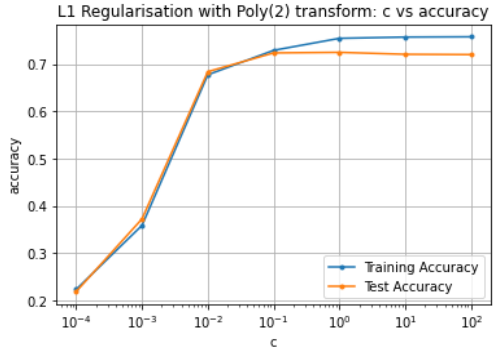
```
C:  0.0001  Train acc:  0.5209247878255779  Test acc:  0.5267220902612827
C:  0.001  Train acc:  0.6262803628914252  Test acc:  0.6318289786223278
C:  0.01  Train acc:  0.6821773485513608  Test acc:  0.6923990498812351
C:  0.1  Train acc:  0.704419081065262  Test acc:  0.7090261282660333
C:  1  Train acc:  0.7085162423178226  Test acc:  0.7125890736342043
C:  10  Train acc:  0.7082235879426397  Test acc:  0.7108076009501187
C:  100  Train acc:  0.7082235879426397  Test acc:  0.7102137767220903
```



The best test accuracy was with a c value of 1, with a training accuracy of 70.85% and a test accuracy of 71.26%.

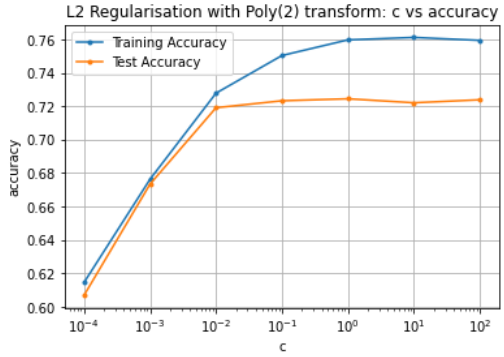
Next, we transformed the features using [scikit-learn's PolynomialFeatures class](). Since we already have a lot of features, we only tested a polynomial feature transformation of degree 2 (this already takes a while to run). With the transformed features, we performed the same code as we did with the normal features. With just the normal logistic regression, we got a test accuracy of 71.73% and a training accuracy of 75.89%. The results with lasso regression is:

```
C:  0.0001  Train acc:  0.22300263388937663  Test acc:  0.21852731591448932
C:  0.001  Train acc:  0.3585016095990635  Test acc:  0.37232779097387175
C:  0.01  Train acc:  0.6774948785484343  Test acc:  0.684085510688836
C:  0.1  Train acc:  0.7292947029558092  Test acc:  0.7238717339667459
C:  1  Train acc:  0.7547556335967223  Test acc:  0.7250593824228029
C:  10  Train acc:  0.7573895229733685  Test acc:  0.7209026128266033
C:  100  Train acc:  0.7579748317237343  Test acc:  0.7203087885985748
```



The results with ridge regression is:

```
C:  0.0001  Train acc:  0.6145741878841089  Test acc:  0.6068883610451307
C:  0.001  Train acc:  0.6763242610477027  Test acc:  0.6733966745843231
C:  0.01  Train acc:  0.7278314310798947  Test acc:  0.7191211401425178
C:  0.1  Train acc:  0.7503658179689786  Test acc:  0.7232779097387173
C:  1  Train acc:  0.7597307579748317  Test acc:  0.7244655581947743
C:  10  Train acc:  0.7611940298507462  Test acc:  0.7220902612826603
C:  100  Train acc:  0.7594381035996488  Test acc:  0.7238717339667459
```



Support Vector Machine

We used the support vector classifier provided by sklearn.svm.SVC. We tested three different kernels: Linear, Polynomial, and Radial Basis Function. Each model also had different parameters to try. For the Linear model, we varied the C parameter. For Poly and RBF models, we varied both C and gamma. A small C allows for some incorrect classifications to maximize the margin, while a large C allows for a smaller margin in order to increase training accuracy. Gamma represents the range of a support vector's influence, with small gamma meaning far range and large gamme meaning close range, as explained by sklearn here. To test many combinations of parameters, we used sklearn.model_selection.GridSearchCV, though the Pipeline only had one step which was the SVC model. Some code was adopted from Jake VanderPlas' SVM tutorial.

LINEAR

We used factors of 10 as the C value, from 10^{-3} to 10^{3}, and found that C >= 1 gave the best overall accuracy. Also, training and testing accuracy were close to each other for all C. To investigate whether the model was equally good at classifying all positions, we used sklearn.metrics.classification_report. The report for C=1 is given below:

```
C= 1 , Train Accuracy= 0.707638279192274
              precision    recall  f1-score   support

           C       0.64      0.69      0.66       492
          PF       0.62      0.59      0.60       673
          PG       0.89      0.89      0.89       741
          SG       0.71      0.70      0.70       749
          SF       0.65      0.65      0.65       762

    accuracy                           0.71      3417
   macro avg       0.70      0.70      0.70      3417
weighted avg       0.71      0.71      0.71      3417

C= 1 , Test Accuracy= 0.7048693586698337
              precision    recall  f1-score   support

           C       0.65      0.67      0.66       230
          PF       0.63      0.60      0.61       337
          PG       0.87      0.90      0.88       365
          SG       0.70      0.68      0.69       384
          SF       0.64      0.65      0.65       368

    accuracy                           0.70      1684
   macro avg       0.70      0.70      0.70      1684
weighted avg       0.70      0.70      0.70      1684
```

The report was similar for other C values. We noticed that the f1 score was significantly higher for point guards while middling for centers and forwards.

POLYNOMIAL

We used C values [1, 5, 10, 50] and gamma values [0.0001, 0.0005, 0.001, 0.005]. GridSearchCV returned that the best parameters were C=50 and gamma=0.005, resulting in this classification report:

```
print(classification_report(y_train,poly_train_pred,lab
```

	precision	recall	f1-score	support
C	0.83	0.44	0.58	492
PF	0.59	0.31	0.40	673
PG	0.98	0.38	0.55	741
SG	0.45	0.92	0.61	749
SF	0.40	0.52	0.45	762
accuracy			0.52	3417
macro avg	0.65	0.51	0.52	3417
weighted avg	0.63	0.52	0.51	3417

```
print(classification_report(y_test,poly_test_pred,label
```

	precision	recall	f1-score	support
C	0.80	0.41	0.54	230
PF	0.57	0.29	0.39	337
PG	0.98	0.43	0.60	365
SG	0.47	0.91	0.62	384
SF	0.38	0.50	0.43	368
accuracy			0.52	1684
macro avg	0.64	0.51	0.52	1684
weighted avg	0.62	0.52	0.52	1684

We noticed that f1 was much lower compared to the linear kernel. Precision and recall were no longer close for each class. Centers and point guards had high precision but low recall, suggesting that the threshold for classifying them was very high, whereas shooting guards had low precision and high recall.

RADIAL BASIS FUNCTION

We tried the same C and gamma values as the polynomial kernel and GridSearchCV returned the same best parameters of C=50 and gamma=0.005. The report is given:

```
print(classification_report(y_train,rbf_train_pred,labe
```

	precision	recall	f1-score	support
C	0.68	0.75	0.71	492
PF	0.66	0.61	0.63	673
PG	0.89	0.90	0.90	741
SG	0.72	0.71	0.72	749
SF	0.67	0.67	0.67	762
accuracy			0.73	3417
macro avg	0.72	0.73	0.73	3417
weighted avg	0.73	0.73	0.73	3417

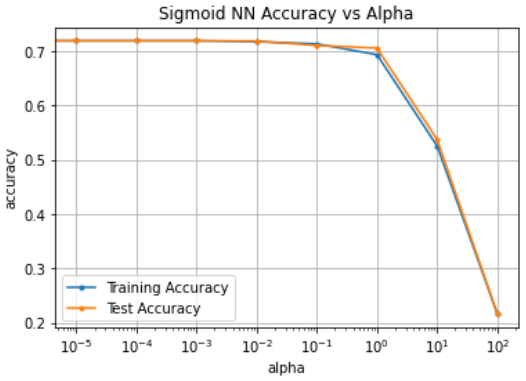
```
print(classification_report(y_test,rbf_test_pred,labels
```

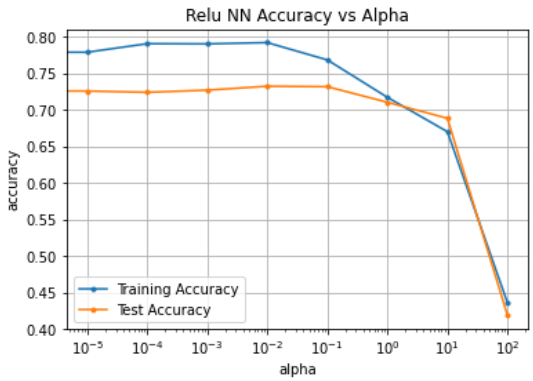
	precision	recall	f1-score	support
C	0.67	0.70	0.69	230
PF	0.65	0.62	0.63	337
PG	0.87	0.90	0.88	365
SG	0.71	0.68	0.70	384
SF	0.64	0.66	0.65	368
accuracy			0.71	1684
macro avg	0.71	0.71	0.71	1684
weighted avg	0.71	0.71	0.71	1684

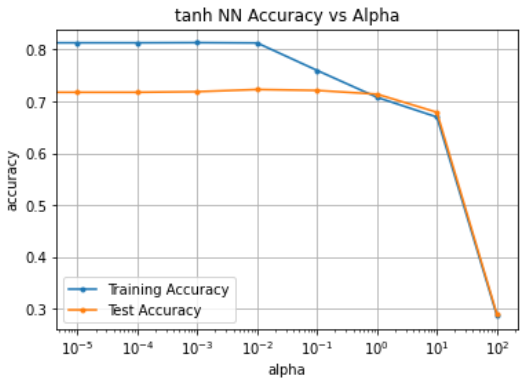
This kernel gave slightly higher accuracy than the linear one, and we see again that precision and recall were best when classifying point guards.

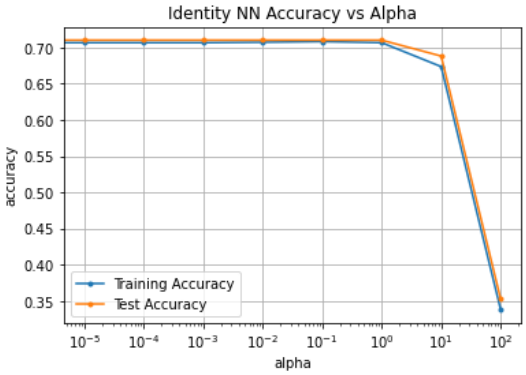
Neural Network

The last model we tried was a neural network. For this, we used scikit-learn's MLPClassifier. The MLPClassifier has a parameter "alpha", which is the L2 penalty parameter (which we thought was the λ value in regularization). To make sure, we did some testing. We made models using four different activation functions: sigmoid, ReLu, tanh, and the identity function, and used the same hidden layer (one layer with 50 nodes). We tried this for different alpha values: [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0], with the 0 being if there was no regularization. We graphed the resulting accuracies with the alpha values:

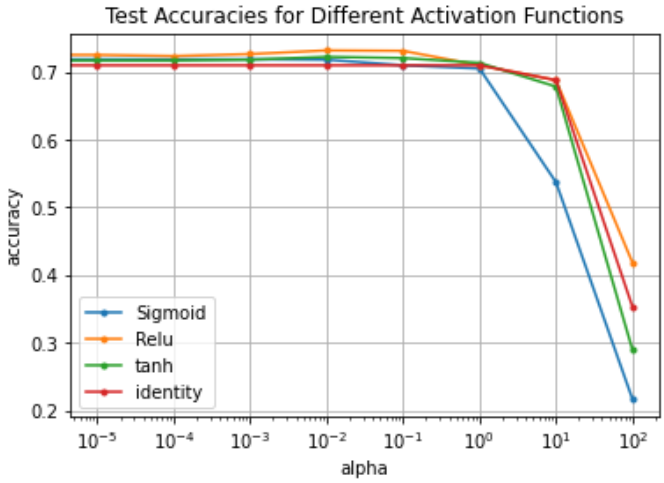








This shows that the alpha value is, in fact, the regularization term. In all the graphs, there are signs of underfitting when alpha is high, and for ReLU and tanh activation functions, there are signs of overfitting when the alpha is small. We also graphed the testing accuracies for all of the activation functions:



The ReLU activation function with alpha = 0.01 seems to have the best testing accuracy at 73.22% and training accuracy 79.19%.

Next, we tried different hidden layer sizes along with the different alphas for each activation function. After some testing with a broad range, we noticed that for each activation function, alpha values of 1 and above always resulted in a lower accuracy, and large hidden layer sizes (i.e. 100, 200) as well as large sizes with multiple layers (i.e. 2 hidden layers with 70 nodes each) also resulted in a lower accuracy for each activation function. So we dropped some of the alpha values and narrowed the range of hidden layer sizes to test. We tested the alpha values [0.1, 0.01, 0.001, 0.0001, 0.00001, 0] and hidden layer sizes [(30), (30,30), (35), (40), (40,40), (45), (50), (50,30), (50,50), (55), (60), (65), (70)], where the hidden layers are represented as a tuple with each element being the number of nodes in one layer (so tuples with one element means one hidden layer of that size, and tuples with two elements mean two hidden layers with the respective sizes). We found that the ReLU function gave the best accuracy overall, with one hidden layer of size 35 and alpha value of 0.001:

```
The best test accuracy for relu is 0.7351543942992874 with hidden layer sizes 35 and alpha value 0.001

Alpha= 0.001 , Train Accuracy= 0.7810945273631841
              precision    recall  f1-score   support

           C       0.80      0.74      0.77       492
          PF       0.71      0.74      0.73       673
          PG       0.91      0.92      0.91       741
          SG       0.77      0.77      0.77       749
          SF       0.72      0.71      0.72       762

    accuracy                           0.78      3417
   macro avg       0.78      0.78      0.78      3417
weighted avg       0.78      0.78      0.78      3417

Alpha= 0.001 , Test Accuracy= 0.7351543942992874
              precision    recall  f1-score   support

           C       0.74      0.64      0.69       230
          PF       0.64      0.69      0.67       337
          PG       0.88      0.93      0.90       365
          SG       0.75      0.70      0.72       384
          SF       0.66      0.68      0.67       368

    accuracy                           0.74      1684
   macro avg       0.73      0.73      0.73      1684
weighted avg       0.74      0.74      0.73      1684
```

The model with the best test accuracy for sigmoid is:

```
The best test accuracy for sigmoid is 0.7280285035629454 with hidden layer sizes (50, 30) and alpha value 0.0001

Alpha= 0.0001 , Train Accuracy= 0.7702663154814164
              precision    recall  f1-score   support

           C       0.82      0.74      0.78       492
          PF       0.70      0.76      0.73       673
          PG       0.91      0.90      0.90       741
          SG       0.75      0.75      0.75       749
          SF       0.70      0.70      0.70       762

    accuracy                           0.77      3417
   macro avg       0.78      0.77      0.77      3417
weighted avg       0.77      0.77      0.77      3417

Alpha= 0.0001 , Test Accuracy= 0.7280285035629454
              precision    recall  f1-score   support

           C       0.76      0.62      0.69       230
          PF       0.65      0.70      0.68       337
          PG       0.88      0.90      0.89       365
          SG       0.73      0.69      0.71       384
          SF       0.64      0.68      0.66       368

    accuracy                           0.73      1684
   macro avg       0.73      0.72      0.72      1684
weighted avg       0.73      0.73      0.73      1684
```

For tanh:

```
The best test accuracy for tanh is 0.7304038004750594 with hidden layer sizes 55 and alpha value 0.1

Alpha= 0.1 , Train Accuracy= 0.7764120573602575
              precision    recall  f1-score   support

           C       0.77      0.78      0.78       492
          PF       0.73      0.72      0.72       673
          PG       0.91      0.91      0.91       741
          SG       0.75      0.77      0.76       749
          SF       0.72      0.70      0.71       762

    accuracy                           0.78      3417
   macro avg       0.78      0.78      0.78      3417
weighted avg       0.78      0.78      0.78      3417

Alpha= 0.1 , Test Accuracy= 0.7304038004750594
              precision    recall  f1-score   support

           C       0.74      0.67      0.70       230
          PF       0.64      0.68      0.66       337
          PG       0.90      0.89      0.90       365
          SG       0.72      0.74      0.73       384
          SF       0.66      0.64      0.65       368

    accuracy                           0.73      1684
   macro avg       0.73      0.72      0.73      1684
weighted avg       0.73      0.73      0.73      1684
```

And for identity:

```
The best test accuracy for identity is 0.7173396674584323 with hidden layer sizes (50, 50) and alpha value 0.01

Alpha= 0.01 , Train Accuracy= 0.7082235879426397
              precision    recall  f1-score   support

           C       0.65      0.70      0.67       492
          PF       0.63      0.58      0.60       673
          PG       0.89      0.88      0.89       741
          SG       0.70      0.71      0.70       749
          SF       0.64      0.66      0.65       762

    accuracy                           0.71      3417
   macro avg       0.70      0.71      0.70      3417
weighted avg       0.71      0.71      0.71      3417

Alpha= 0.01 , Test Accuracy= 0.7173396674584323
              precision    recall  f1-score   support

           C       0.68      0.67      0.68       230
          PF       0.64      0.60      0.62       337
          PG       0.90      0.89      0.89       365
          SG       0.71      0.71      0.71       384
          SF       0.64      0.69      0.66       368

    accuracy                           0.72      1684
   macro avg       0.71      0.71      0.71      1684
weighted avg       0.72      0.72      0.72      1684
```

Notice that for all of these, the f1-score, precision and recall for point guards are much higher than the others.

We also printed all the training and test accuracies for all the different alpha values and hidden layer sizes in a table. The tables for all the activation functions can be found in the table of results section, and the classification report along with the table can be found in the code linked above.

TABLE OF RESULTS

Model	**Regularization/ Transformation/ Kernel/ Activation Function**	**Best parameters**	**Training accuracy**	**Testing accuracy**
Logistic Regression	None		0.7114	0.7084
Logistic Regression	L1	C=1	0.7088	0.7120
Logistic Regression	L2	C=1	0.7085	0.7126
Logistic Regression	Poly(2)		0.7583	0.7185
Logistic Regression	L1, Poly(2)	C=1	0.7548	0.7251
Logistic Regression	L2, Poly(2)	C=1	0.7597	0.7245
SVM	Linear	C=1	0.7076	0.7049
SVM	Polynomial	C=50 Gamma=0.005	0.41	0.41
SVM	Radial Basis Function	C=50 Gamma=0.005	0.73	0.71
Neural Network	Sigmoid	Alpha = 0.0001 HL Size = (50, 30)	0.7703	0.728
Neural Network	ReLU	Alpha = 0.001 HL Size = (35)	0.7811	0.7352
Neural Network	Tanh	Alpha = 0.1 HL Size = (55)	0.7764	0.7304
Neural Network	Identity	Alpha = 0.01 HL Size = (50,50)	0.7082	0.7173

For neural networking, we printed the accuracy for all the different alpha values and hidden layer sizes we tested in a table. In the tables below, the columns are the alpha values with the labels on top, and the rows are the hidden layer sizes with the labels to the right.

Training Accuracies for Sigmoid Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.7106	0.7287	0.7299	0.7413	0.7413	0.7413	30
0.712	0.717	0.7296	0.7416	0.7419	0.7416	(30, 30)
0.7123	0.7278	0.782	0.7834	0.7831	0.7831	35
0.7123	0.722	0.7217	0.7229	0.7229	0.7229	40
0.7106	0.7211	0.7229	0.7234	0.7234	0.7234	(40, 40)
0.7114	0.727	0.7369	0.7413	0.7413	0.7413	45
0.7132	0.7173	0.7191	0.7191	0.7191	0.7191	50
0.7138	0.7167	0.7711	0.7703	0.7706	0.7706	(50, 30)
0.7155	0.7158	0.7152	0.7152	0.7152	0.7152	(50, 50)
0.7114	0.7179	0.7199	0.7202	0.7202	0.7202	55
0.7109	0.7158	0.7624	0.7638	0.7641	0.7641	60
0.7088	0.7164	0.7346	0.7337	0.734	0.734	65
0.7112	0.7193	0.7202	0.8197	0.8186	0.818	70

Test Accuracies for Sigmoid Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.7096	0.712	0.7138	0.7221	0.7221	0.7221	30
0.7084	0.7167	0.7185	0.7257	0.7257	0.7257	(30, 30)
0.7102	0.7173	0.7227	0.7268	0.7262	0.7257	35
0.7114	0.7144	0.7156	0.715	0.715	0.715	40
0.7144	0.7167	0.7162	0.7156	0.7156	0.7156	(40, 40)
0.7102	0.7138	0.7251	0.7251	0.7251	0.7251	45
0.7102	0.7185	0.7191	0.7191	0.7191	0.7191	50
0.715	0.712	0.7257	0.728	0.728	0.728	(50, 30)
0.709	0.7162	0.7167	0.7173	0.7173	0.7173	(50, 50)
0.7078	0.7162	0.7173	0.7179	0.7179	0.7179	55
0.712	0.712	0.7209	0.7215	0.7215	0.7215	60
0.7126	0.715	0.7251	0.7262	0.7257	0.7257	65
0.7055	0.7197	0.7173	0.7197	0.7173	0.7173	70

Training Accuracies for ReLu Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.748	0.7574	0.7577	0.7521	0.7539	0.7562	30
0.7884	0.8206	0.8294	0.8352	0.8344	0.8355	(30, 30)
0.7574	0.7694	0.7811	0.7805	0.7796	0.7782	35
0.7571	0.7635	0.7726	0.7811	0.7811	0.779	40
0.8197	0.8742	0.8859	0.9075	0.8847	0.8461	(40, 40)
0.7682	0.7872	0.791	0.7869	0.7869	0.7869	45
0.7682	0.7919	0.7902	0.7905	0.7788	0.7899	50
0.8045	0.8689	0.861	0.8984	0.8821	0.9023	(50, 30)
0.8411	0.897	0.8964	0.9046	0.897	0.8955	(50, 50)
0.7764	0.7934	0.8001	0.7937	0.8036	0.8019	55
0.7711	0.794	0.8016	0.8057	0.8036	0.8042	60
0.7673	0.7954	0.7972	0.8229	0.8183	0.8165	65
0.7796	0.8033	0.8033	0.8068	0.8068	0.8063	70

Test Accuracies for ReLu Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.7221	0.7274	0.7251	0.7257	0.7257	0.7245	30
0.7061	0.7108	0.6989	0.7055	0.7031	0.6966	(30, 30)
0.7322	0.7245	0.7352	0.7298	0.7251	0.734	35
0.7239	0.7203	0.7215	0.7197	0.7221	0.7227	40
0.709	0.6936	0.6841	0.6894	0.6835	0.7013	(40, 40)
0.7251	0.7191	0.7173	0.7185	0.7203	0.7203	45
0.7316	0.7322	0.7268	0.7239	0.7257	0.7245	50
0.7114	0.6995	0.6977	0.696	0.7078	0.6954	(50, 30)
0.7138	0.6966	0.6942	0.6995	0.6859	0.6989	(50, 50)
0.7257	0.7203	0.7132	0.7251	0.715	0.7144	55
0.7268	0.7221	0.7179	0.7185	0.7162	0.7179	60
0.7286	0.7221	0.7286	0.7185	0.7274	0.7162	65
0.7262	0.7209	0.7286	0.7286	0.7221	0.7227	70

Training Accuracies for Tanh Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.7685	0.7811	0.7823	0.7829	0.7831	0.7831	30
0.8308	0.9353	0.9388	0.9523	0.9529	0.9532	(30, 30)
0.7653	0.7925	0.7934	0.7943	0.7943	0.7943	35
0.7656	0.7963	0.8016	0.8025	0.8025	0.8025	40
0.8832	0.9971	1.0	1.0	1.0	1.0	(40, 40)
0.7711	0.8142	0.8171	0.8171	0.8171	0.8171	45
0.7594	0.8124	0.813	0.8127	0.8127	0.8127	50
0.8733	0.9927	0.998	0.9985	0.9988	0.9988	(50, 30)
0.8938	0.9991	1.0	1.0	1.0	1.0	(50, 50)
0.7764	0.8247	0.8493	0.8493	0.8493	0.8493	55
0.7858	0.837	0.8396	0.839	0.8393	0.8393	60
0.7673	0.8227	0.8285	0.8288	0.8288	0.8291	65
0.7711	0.854	0.8639	0.8642	0.8642	0.8642	70

Test Accuracies for Tanh Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.7262	0.7251	0.7215	0.7221	0.7221	0.7221	30
0.7274	0.6776	0.6853	0.6758	0.6752	0.6758	(30, 30)
0.7274	0.7257	0.7286	0.7292	0.7292	0.7292	35
0.7173	0.7031	0.7055	0.7055	0.7055	0.7055	40
0.712	0.6776	0.6586	0.6568	0.6574	0.6574	(40, 40)
0.7239	0.7162	0.7179	0.7179	0.7179	0.7179	45
0.7209	0.7227	0.7185	0.7173	0.7173	0.7173	50
0.7215	0.6692	0.6639	0.6657	0.6651	0.6657	(50, 30)
0.7221	0.6764	0.6841	0.6847	0.6847	0.6847	(50, 50)
0.7304	0.7197	0.6977	0.6983	0.6983	0.6983	55
0.728	0.7156	0.7126	0.7138	0.7138	0.7138	60
0.7251	0.7114	0.7138	0.7132	0.7132	0.7132	65
0.7179	0.7019	0.6882	0.6882	0.6882	0.6882	70

Training Accuracies for Identity Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.7079	0.7071	0.7071	0.7071	0.7071	0.7071	30
0.71	0.7044	0.7047	0.7047	0.7047	0.7047	(30, 30)
0.7059	0.7094	0.7091	0.7091	0.7091	0.7091	35
0.7068	0.7079	0.7079	0.7079	0.7079	0.7079	40
0.7079	0.7085	0.7085	0.7085	0.7085	0.7085	(40, 40)
0.7073	0.7073	0.7071	0.7071	0.7071	0.7071	45
0.7082	0.7073	0.7068	0.7068	0.7068	0.7068	50
0.7097	0.7088	0.7088	0.7091	0.7091	0.7091	(50, 30)
0.7071	0.7082	0.7079	0.7079	0.7079	0.7079	(50, 50)
0.7117	0.7106	0.7103	0.7103	0.7103	0.7103	55
0.7094	0.7079	0.7085	0.7085	0.7085	0.7085	60
0.7073	0.7079	0.7082	0.7082	0.7082	0.7082	65
0.7106	0.7097	0.7094	0.7094	0.7094	0.7094	70

Test Accuracies for Identity Activation Function:

0.1	0.01	0.001	0.0001	1e-05	0	
0.709	0.7078	0.7078	0.7078	0.7078	0.7078	30
0.7084	0.7102	0.7102	0.7102	0.7102	0.7102	(30, 30)
0.7102	0.7144	0.7144	0.7144	0.7144	0.7144	35
0.7132	0.712	0.7126	0.7126	0.7126	0.7126	40
0.7114	0.7108	0.7108	0.7108	0.7108	0.7108	(40, 40)
0.709	0.7067	0.7067	0.7067	0.7067	0.7067	45
0.7102	0.7102	0.7102	0.7102	0.7102	0.7102	50
0.7144	0.7138	0.7138	0.7138	0.7138	0.7138	(50, 30)
0.7132	0.7173	0.7173	0.7173	0.7173	0.7173	(50, 50)
0.715	0.7144	0.7144	0.7144	0.7144	0.7144	55
0.7144	0.7138	0.7132	0.7132	0.7132	0.7132	60
0.709	0.7096	0.7096	0.7096	0.7096	0.7096	65
0.7108	0.7138	0.7138	0.7138	0.7138	0.7138	70

CONCLUSION

Logistic Regression

The best Logistic Regression model used penalty=L2 and C=1 with polynomially transformed data (d=2). However, the accuracy difference between this model and its L1 counterpart was minimal. This model also had the greatest difference between training and testing accuracy, but at only 3%, overfitting was not much of an issue. Overfitting was not observed in any of the logistic regression models, which could be owed to the large dataset available to us. With over 3400 training samples, it's likely that however we split the dataset, the training data would be a good representation of the truth, so variance is quite low.

Support Vector Machine

For SVM, the best model uses the radial basis function kernel with C=50 and Gamma=0.005. The difference between the training and testing accuracy here is only 2%, lower than most of the other models we tried, so there is probably little to no overfitting. The same is true for the other SVM models. However, both the training and test accuracy for the polynomial kernel was very small. We aren't exactly sure why this is happening, perhaps mapping the data to a higher degree polynomial made it difficult to find a good hyperplane as a decision boundary.

Neural Network

Out of all the models we tested with neural networks, the model that gave the best test accuracy uses the ReLU activation function with an alpha value of 0.001 and one hidden layer of size 35. The difference between the training error and test error was around 4.6%,

which is slightly higher relative to some of the other models we tested, but does not seem to have overfit the data too much. When the hidden layer is set to the same value and we tested for the accuracies for different alpha values, every graph showed signs of underfitting when the alpha was large, but only the models for ReLU and tanh seemed to have overfit when the alpha is close to zero.

Overall

The coefficients/weights of the logistic regression model give us many clues about the different basketball positions. For each position, a positive coefficient corresponds to a strength, since a large value for that feature increases the likelihood of that class. On the other hand, a negative coefficient indicates that the feature predicts against that class. Some of the features that had a large impact (the absolute value of the weight is relatively large) on the probability for a position are:

Feature	Positive Impact	Negative Impact
Age	C, PF	PG, SG
TS%	SF	PF
3PAr	PG	C, SF
TRB%	C, PF, SF	PG, SG
AST%	PG	C, PF, SF
STL%	PG, SG	C, PF
BLK%	C, PF	PG, SG
USG%	PF, SF, SG	PG
FG%	C	SG, SF
3P%	SF	PG

2P%	SF	C
FT%	PG	PF, SF

Based on the coefficients, as well as the f1 scores for individual classes given by the classification reports for both SVM and NN, one might say that point guards are the most unique or specialized class. PG coefficients for rebounds, assists, steals, and blocks were among the highest across the matrix, and the model's precision and recall were highest for PG. Conversely, the small forward position had small coefficients for many features, and in fact was the only class with all coefficients between -1 and 1. This suggests that the SF position is less specialized and in the middle of all the other positions. Similarity between classes could also be why the f1 scores were fair but not great for non-PG positions. Looking at the coefficients again, centers and power forwards had the same strengths of rebounding and blocking, and the same weaknesses of assists and steals. The only feature that isn't directly related to game performance is Age. We did not have a strong hypothesis about Age, but included it in case any trends did exist. We found that the Age coefficients were not insignificant, being positive for centers and power forwards, but negative for point guards and shooting guards. A possible explanation is that the former two positions rely less on youth and athleticism, whereas the latter two do in fact require athleticism or perhaps dexterity. Yet, some coefficients were surprising, especially when compared to the original data. Centers have a negative 2P% coefficient, while their FG% coefficient is the highest of the classes. Considering they also have a negative 3PAr coefficient, we would assume that nearly all of their field goal attempts are for 2 points, and their FG% and 2P% should be similar. Checking the original dataset shows that

centers in fact have the highest average 2P%, so we are unsure why the model assigned them a negative coefficient.

Possible Improvements or Future Work

One thing we could do that may improve our understanding of how the different positions relate to a player's stats and to each other is to do unsupervised machine learning. We may be able to better visualize how the positions are related to each other (i.e. perhaps some positions may be more similar to other positions), which stats are more important for certain positions (perhaps a position makes a lot of three point shots, so those players would need to have a high ratio of making three pointers), or maybe find something entirely new or unexpected.

Another thing we could do is try to perform other feature transformations. Because we already started out with a lot of features, just doing a polynomial transformation of degree 2 took a long time to run. If we had more time, we could try even higher degree transformations. Doing cross validation could also help improve our accuracy, as it decreases both bias and variance.

We could also do more preprocessing. Our data for the center position is slightly lower than the other positions, so we could maybe randomly drop some data until there is the same amount of data for each position. We also noticed that dropna() was having some unexpected results. When we tried fillna() instead, the center class gained more samples. We realized that many centers had a null value for 3P% because they would go an entire season without attempting one. That makes those samples quite specialized, but they were being removed from the data by dropna(). In fact, we realized rows before 1979

were missing 3-point data because the 3-point field goal did not even exist yet. This resulted in a lot of the data being from recent games, which might introduce more bias. If we were to continue this work, we might investigate differences across time, since players and strategies are constantly evolving. One modern idea is “positionless basketball,” which encourages more well rounded players. Would it be harder to classify on recent data versus old data, where positions were more traditional?