Data Preprocessing

Here is what the original data looked like.

	%_of_cap	Pos	Age	G	MP	PER	TS%	3PAr	FTr	TRB%	AST%	STL%	BLK%	TOV%	USG%	WS	BPM
0	0.113043	1	26	57	2029	18.6	0.535	0.324	0.164	3.9	34.7	1.6	0.1	10.1	25.1	5.5	1.1
1	0.015652	3	31	53	516	9.6	0.489	0.055	0.133	10.6	9.2	0.9	1.3	13.4	17.7	0.2	-5.2
2	0.021304	1	22	60	560	8.7	0.506	0.426	0.161	4.2	31.6	2.5	0.3	29.1	18.8	0.2	-4.7
3	0.008696	2	23	41	362	7.8	0.447	0.314	0.343	4.1	20.8	3.0	1.1	22.0	19.0	-0.3	-6.1
4	0.015217	5	23	73	1614	11.8	0.518	0.008	0.289	10.8	6.5	1.8	2.4	11.9	13.7	1.7	-1.8

Some preprocessing that I needed to do was to group '%_of_cap' by ranges in order to make this a classification problem and to one-hot encode my categorical feature 'Pos'. I used pandas get_dummies to one-hot encode my categorical feature. To group my data, I used NBA minimum (10%) and maximum (25%) of salary cap allowed to obtain my first and last groups, then segmented what was in between.Finally I removed the %_of_cap and Pos column, and split my data into feature and target variables. The data was rescaled using MinMaxScaler.

Machine Learning Models

5 Models were used: Decision Tree, Random Forest, XGBoost, AdaBoost and SVM.

For the model selection process, I created a pipeline that combines the preprocessing MinMaxScaler with a classifier dictionary containing all the models selected with default parameters. I looped the pipeline with KFold cross-validation to measure model performance.

	classifier	mean_fit_time	mean_test_score	std_test_score	mean_train_score
0	DecisionTree	0.095745	0.666398	0.007840	1.000000
1	RandomForest	0.174659	0.745540	0.009935	0.983501
2	SVC	1.227508	0.730651	0.011307	0.731388
3	AdaBoost	0.874373	0.747150	0.009784	0.758417
4	XGBoost	5.135772	0.758417	0.011314	0.793427

The following metrics were used to evaluate our models:

1. Average test score
    a. The average test score is our accuracy score. This tells us which model performed well on the test data. Decision Tree scored the lowest at 0.667. The other 4 classifiers were very similar with Random Forest edging out as the top model performer in test scores.

2. Average train score
    a. The average train score tells us how well the model performed on trained data set. When comparing the train score and the test score, we can get an idea if our model overfits on the training data, that is it predicts very well on the train data but not as well on the test data, by looking at the difference between train scores and test scores. The larger the difference, the more your model overfits. We can that Decision Tree and Random Forest Classifier models overfit on the training data by a larger margin than the other models.
3. Test score standard deviation
    a. The standard deviation of the test score lets us know the variance of our models. We are looking for models that have a low variance, which will tell us if the prediction we obtained on test is not by chance and that our model will perform similarly on all test sets. SVC and XGBoost have the largest variance.
4. Average fit time
    a. This metric is the average time, in seconds, it takes to fit the model to the data. I want my model to run fairly quick. XGBoost and SVC take the longest to fit, while Decision Tree is the fastest.

The models that performed well enough to dive deeper into were AdaBoost and Random Forest Classifier. AdaBoost Classifier had a high accuracy score, did not overfit, had a low variance, but it does take longer to fit. Random Forest Classifier had a high accuracy score and takes less time to fit but the model does overfit on the training model.

Hyperparameter Tuning

I used GridSeachCV to find optimal parameters.

AdaBoost Tuning:

The hyperparameters that were tuned were learning_rate and n_estimators. The best estimator used 0.1 for the learning rate and 1000 for n_estimators. This lead to an improvement in accuracy score at 0.764

```
ada_predict = pipe_ada.predict(X_test)
print('Model Accuracy: ', accuracy_score(y_test, ada_predict))
print('Training Score: ' , pipe_ada.score(X_train, y_train))
print('Best Parameters: ', pipe_ada.steps[1][1].best_params_)
```

```
Model Accuracy:  0.7637826961770624
Training Score:  0.7691482226693495
Best Parameters:  {'learning_rate': 0.1, 'n_estimators': 1000, 'random_state': 34}
```

```
print(confusion_matrix(y_test, ada_predict))
```

```
[[1762   33    2   11   14]
 [ 192   39    8    4   15]
 [  84   38   11   10   28]
 [  36   15    9   14   22]
 [  18   23    6   19   72]]
```

```
print(classification_report(y_test, ada_predict))
```

```
              precision    recall  f1-score   support

       0-10%       0.84      0.97      0.90      1822
     10%-15%       0.26      0.15      0.19       258
     15%-20%       0.31      0.06      0.11       171
     20%-25%       0.24      0.15      0.18        96
       25%<=       0.48      0.52      0.50       138

   micro avg       0.76      0.76      0.76      2485
   macro avg       0.43      0.37      0.38      2485
weighted avg       0.70      0.76      0.72      2485
```

Random Forest Tuning:

In this classifier I tuned max_features, n_estimators, and max_depth to help counteract the model overfit. The best parameters found was ‘auto’ for max_features, 2000 for n_estimators, and 8 for max_depth. The model accuracy improved to 0.763 and brought the training score down to 0.795. That’s a significant improvement from the default method.

```
rf_predict = pipe_rf.predict(X_test)

print('Model Accuracy: ', accuracy_score(y_test, rf_predict))
print('Training Score: ' , pipe_rf.score(X_train, y_train))
print('Best Parameters: ', pipe_rf.steps[1][1].best_params_)
```

```
Model Accuracy:  0.7625754527162978
Training Score:  0.794634473507713
Best Parameters:  {'max_depth': 8, 'max_features': 'auto', 'n_estimators': 2000, 'random_state': 34}
```

```
print(confusion_matrix(y_test, rf_predict))
```

```
[[1803    6    1    1   11]
 [ 226   16    5    0   11]
 [ 126   10    4    1   30]
 [  63    5    0    0   28]
 [  50   10    5    1   72]]
```

```
print(classification_report(y_test, rf_predict))
```

```
              precision    recall  f1-score   support

       0-10%       0.79      0.99      0.88      1822
     10%-15%       0.34      0.06      0.10       258
     15%-20%       0.27      0.02      0.04       171
     20%-25%       0.00      0.00      0.00        96
       25%<=       0.47      0.52      0.50       138

   micro avg       0.76      0.76      0.76      2485
   macro avg       0.38      0.32      0.31      2485
weighted avg       0.66      0.76      0.69      2485
```

In this situation, I want the end user to be able to see who is getting over and undervalued for the kind of stats they bring into a game. For an NBA teams front office, potentially finding players who can give the kind of stats that warrant <20% of the salary cap but currently getting paid lower and vice versa. Or for NBA players to gauge, with their current stats, whether they are getting categorized at a higher or lower percentage of the salary cap than what they currently are being paid. For this reason, I placed more importance on recall value. AdaBoost Classifier had the lower variance and higher recall and precision values