FINAL RESEARCH ASSIGNMENT

Brett Mele

MSDS458 Northwestern University

June 4, 2023

Abstract

NFL play prediction is a popular application of now widely available play-by-play data. Better anticipating the next play type in a game can be a tremendous edge for NFL teams. In this paper we introduce methods for capturing sequential dependencies across plays within games to improve on prior, static models that rely solely on the current game state to predict the next play. We propose two models using LSTM architectures – one for predicting field goals, punts, passes and runs on all plays and another for predicting run vs pass on non-fourth down plays. Both models improve on existing methods and better handle the imbalance of passing and running plays.

1. Introduction

In the National Football League (NFL), anticipating what play an opponent may run on offense can make a substantial impact on the eventual outcome of the game. Having the right play called to defend a passing play, and having defenders anticipating a pass, will make it increasingly difficult for the offense to find success. Conversely, using a play to defend the run when the offense actually passes the ball gives the offense a significant advantage. Thus coaches who can better anticipate what the opposing offense will do on the next play can give their team an edge on every play call throughout the game, which in aggregate can have a meaningful impact on the eventual game outcome.

In an NFL game, there are four primary play types that an offense can call during standard play calling situations – pass, run, kick and punt (we refer to standard play calling situations as all situations excluding point-after attempts and kickoffs). Passes and run plays typically occur on downs one, two and three. On fourth downs, the offense must decide between attempting to kick

a field goal to score points, calling a play to gain a first down (or score a touchdown), or punt the ball away and give possession to the opponent. Pass plays gain more yardage and are more valuable on average than run plays. But run plays can be particularly effective when only a few yards are needed to score or gain a first down, or when the defense is anticipating a pass and changes defensive personnel and strategy accordingly. Field goals tend to be attempted when the expected value of the kick is larger than the expected value of attempting to gain a first down or punt the ball away. Punts tend to only occur when there is a low chance of gaining a first down or making a field goal.

Due to the uncertainty surrounding what type of play will be called in a given situation, model based approaches have been developed to help predict play types. These approaches arose in large part because of the increase in access to public play-by-play data. Starting with nflscrapR (Horowitz, Yurko & Ventura 2017), which has now become nflfastR (Carl and Baldwin 2023), play by play data is available for all NFL games dating back to 1999.

In this manuscript, we outline a novel approach to NFL play prediction that improves on existing frameworks. Our model deviates from earlier approaches by introducing knowledge about prior play sequences with the use of recurrent neural networks (RNN). To our knowledge, all public play prediction models rely on the game state alone (time remaining, down, yards to go for first down, yardline, etc.) to predict the next play in a game. However, we posit that the history of play calls throughout the game can provide a signal as to what a team may do later on in a game. For example, if a team has had success running earlier in the game, they may run in a late-game situation even if a pass is typically expected in that situation.

To build our play model we experiment with multivariate RNN architectures including but not limited to long-short term memory (LSTM), gated recurrent unit (GRU) and 1-dimensional convolutional (1D-CNN) networks. In subsequent sections we discuss experimental design, comment on architectural choices, present the results of experiments and discuss model performance. After investigating the impact of preprocessing and architecture choices, we discuss the strengths and weaknesses of our approach. Ultimately we present two models, one for predicting play types in standard game situations and one for predicting "dropbacks" (run or pass) in standard situations excluding fourth downs, each of which improve on existing frameworks.

2. Literature Review

Muzumdar, Kharosekar and Jain (2017) explored multiple machine learning and regression approaches to play prediction using data from 2015. Patel (2020) presented an approach using an xgboost model to predict play types given the game state. He suggests LSTM as a possible extension to his work to include information about previous play calls. Baldwin (2021) has perhaps the most well known play prediction model given its inclusion in the nflfastR dataverse. He leveraged game-state features within an xgboost framework to predict the likelihood of a pass or run in non-kicking situations.

3. Data & Methods

We leverage nflfastR to gather play-by-play data from 2012 through the 2022 season. The resulting dataset contains 536,440 plays across 2,974 games. Each play has 372 features which include play and game metadata, information about the game state at the beginning and end of the play, player information, model predictions and more. A complete data dictionary is available

on the nflreadr website. After collecting the data, we filter the dataset to include only regular season games within each season. We also exclude any non-standard plays like kickoffs, extra points and two-point conversion attempts.

For the purposes of our model, we engineer features along with game state information provided by nflfastR. The first set of our engineered features attempts to provide information about the quality of the team in possession of the ball (offensive team), tendencies of the offensive team, and quality of the defensive team. These features are included in our data with prefix “pt” for possession team and “dt” for defensive team. To aggregate these features and apply them to individual plays, we take cumulative rates in each season up to but excluding the game in question. The idea is to allow the model to learn about team ability while not introducing information about the current game. Features include rates of pass plays overall and by situation, and efficiency metrics including expected points added (EPA) per play and win probability added (WPA) per play.

The second set of engineered features capture the quality of each quarterback within the dataset. Quarterback, relative to other positions, is perhaps the most important in all of professional sports. Independent of team quality, the quality of the quarterback on the offensive team during each play can significantly impact the choice of play call. To capture quarterback quality we compute WPA and EPA per play similar to the method above, along with cumulative pass attempts in a season. For WPA and EPA per play, we take additional steps to account for small sample size and variance across seasons by using a three-year weighted average of each metric as a Bayesian empirical prior for the cumulative rates in each season.

After calculating team and quarterback quality metrics, we one hot encode categorical features and fill in missing numeric data with either a group mean or zero depending on the specific variable. We then create a column for our target variable play type which initially takes 6 values: field goal, pass, punt, run left, run middle, run right. After model experimentation we remove direction from run plays, resulting in 4 target classes.

To prepare data for modeling we first split data into train, validation and test sets. To do so we opt for the following pseudo-random approach rather than a typical random split. The entire 2022 season is held out as the test set. Games from 2016-2021 are then used as training data. To split these into training and validation sets, all games from odd weeks are used in the training set along with games from weeks 4, 8, 12 and 16 from even seasons and games from weeks 2, 6, 10 and 14 from odd seasons. The rest of the games in the training data are used for validation. We use this approach to better control the amount of data used in modeling and to ensure that season and week level biases (eg. differences in play calling early in the season versus late) are not introduced into the model. This results in 3,104 training games and 748 validation games, which amounts to about a 76/24 split. Note a game in this case is twice the number of actual games because there are two sets of contiguous sequences of plays within a game, one for each team. 542 games are included in the hold-out set.

Once split we preprocess and restructure the data to use in our RNN architecture. First we use categorical encoding for the target variable play type and scale all numeric features from 0 to 1. Once encoded, we create sequences of plays to use in our time series model. To do so we create custom functions to group plays into games (again where game is defined as a set of plays for a team in a game) and pad all sequences to the maximum play-length in the entire dataset. This ensures that sequences that are independent of each other (i.e. occur in different games) are not

linked, and that all sequences are of the same length. In other words, the model will only learn from sequences of plays that occur in individual games. To create batched train, validation and test datasets of play sequences we use the timeseries_dataset_from_array utility from the Keras package. We batch individual games first and then concatenate the individual batched datasets together to again ensure that sequences within each batch are from the same game.

The goal of model experimentation was to develop an optimal model for play type prediction as determined by our loss function (sparse categorical cross entropy). We choose loss to evaluate our models due to class imbalance. Pass plays are almost four times as prevalent as the next highest class when using 6 targets (figure 1) and 9 times as prevalent as field goals and punts. Even when removing run direction from our targets, pass plays still occur almost 1.5 times as often as run plays (figure 2). We conducted a total of 33 model experiments, consisting of multiple variations of RNN architecture. LSTM, GRU and 1D-CNN layers were our main focus to capture sequential patterns. Within each model type we adjust the number of hidden units, number of layers, amount and type of regularization, batch size, input sequence length and sequence shuffling within batches. Additionally, because our data is multivariate we test a few different combinations of play-level features included at each step in a sequence. Each model regardless of type includes a making layer to ignore padded sequences, ends with a dense softmax layer with nodes equal to the number of target classes, is not stateful (to ensure independence across batches), and is compiled using the Adam optimizer. Model callbacks include early stopping during training and specification that returns only the model from the best training iteration.

Data collection and feature engineering were conducted in R. Model implementation in Python leverages the Keras package via Tensorflow. Matplotlib and Seaborn are used to visualize model

results while Scikit-learn was used in reporting model performance. A full list of Python packages and R libraries can be found in the accompanying code repositories.

4. Results

4.1 Model Experimentation[1]

Experimentation began with a RNN with a single LSTM layer and 16 recurrent nodes to predict the 6 class play type. In the initial models we used 58 features corresponding to game state, team quality and quarterback quality. This amounted to somewhat of a kitchen-sink approach as we wanted to get an idea of what features were important and which gave the model a hard time. Batch size was 64 with sequences of length 15. The batch size was chosen because the average length of a game in the training dataset was about 68 plays. By using a batch size close to the average number of games, most games would be processed as one batch. Training data for this model included just the 2019 season, as we wanted to simply establish a baseline before introducing more data.

Training results for the first experiment can be seen in figure 4. The model began to overfit rather quickly as seen in the loss curve. Test loss was about the same as validation loss at 0.847. Visualizing results with a confusion matrix (figure 5), we see the model struggled with class imbalance as it predicted mostly pass plays if the play in question was not a field goal or punt.

To aid in feature selection for subsequent models, a random forest classifier was constructed to predict the modeled play type from flattened sequences. Then we examined feature importances across sequence timesteps to try to understand what features the model was using to make

[1] Model information and performance metrics for all experiments included in the appendix

predictions. This plot for the last timestep in each sequence is shown in figure 8. At least in the last time step, qb_epa, epa, wp (win probability) and wpa were the most important features, while binary indicators like quarter_end and timeout were not important.

Subsequent experiments with LSTM layers with two additional years of training data yielded little to no performance improvement. Adding dropout and recurrent dropout to the LSTM architecture did decrease test loss, but only marginally. Class imbalance was still a major issue as well (figure 7).

The next set of experiments involved GRU layers using training data from 2020-2021 to speed up training time. We tested shorter sequence lengths (10) and longer batch sizes (100), along with more recurrent nodes. None of the models significantly outperformed the LSTMs in terms of validation loss, so we did not bother with model evaluation on the test set.

Returning back to the LSTM architecture, we next conducted a number of tests with varying recurrent nodes and number of recurrent layers, but keeping the same batch size and sequence length from the previous GRU models. We also dropped a few unnecessary features regarding the location of the pass on the play (left, deep, right). No approach, whether stacking layers, using bidirectional LSTMs or increasing nodes made much of a performance difference. The best model in this set of experiments was a one-layer bidirectional LSTM with 64 recurrent nodes. However, the model was still overfitting to pass plays on the training data, so we did not evaluate any of these models.

In an attempt to figure out the issue of class imbalance, we ran a number of further tests with class weights, shuffled sequences and varied batch sizes, sequence lengths, and number of features. Class weights and shuffled sequences in particular turned out to be poor approaches.

With class weights, the model ended up overfitting to minority classes (figure 8). With shuffled sequences, our assumption was that batches would be shuffled so that subsequent batches may not be in the same order they were in the training data. We realized that shuffling was occurring within batches, meaning the temporal relationship between sequences was broken, which produces nonsensical results with multiple models with validation loss greater than 1. After multiple failed experiments we decided to switch from 6 target classes to 4 by removing the run direction classes. This change, along with the removal (redundant variables like fg_attempt) and addition of several features (additional game state and team quality variables like yards_gained, drive_start, dt_epa_play) did seem to improve model performance, other issues notwithstanding. The best model with shuffled sequences, improved input features, and 4 class prediction (experiment 24) yielded test loss of 0.619. Model performance plots can be seen in figures 9 and 10. The model still predicted far too many field goals and almost never predicted punts correctly.

In our final set of experiments, we removed sequence shuffling when creating the timeseries datasets to vastly improved results. We also increased batch sizes to 87 with sequence length of 15, allowing the majority of games to be processed in entire batches as the vast majority of games are less than 90 plays. When testing models with training data from just 2021, the best performing model (experiment 28) was a bidirectional LSTM with 64 nodes, 20% recurrent dropout and a 30% dropout layer. Test loss for this model was 0.589. The model converged nicely (figure 11) with a much better class balance (figure 12 and 13), though precision was much higher for field goals and punts than run and pass plays.

After identifying this best performing model, we introduced an additional year of training data, reduced the number of nodes to combat overfitting, and allowed the model to run for more epochs. These changes resulted in a test loss of 0.514 (74% accuracy) which was a significant

improvement. Classification results for this model are included in figures 14 and 15. The F1 score on both passing field goals and punts was over 90, indicating the model did a very good job at learning when those classes may occur. When predicting run versus pass, the model performed worse, with F1 scores of 0.70 and 0.53 respectively.

Given other play prediction models focus solely on pass versus run plays, we trained an additional binary classification model on non-fourth down plays to predict pass versus run. This model included class weights to correct for the slight imbalance between pass and run plays. We achieved 74% accuracy, with F1 scores of 0.80 on passing plays and 0.61 on run plays (figure 16).

4.2 Discussion

Our initial goal of predicting 6 classes, including the direction of running plays, failed. The models were not able to handle the intense class imbalance and had no understanding that each of the types of running plays were related. However, when we switched to four and two class predictions, the models significantly improved.

The best architectures ended up being bidirectional LSTMs. Models performed similarly regardless of the number of nodes. Larger node sizes tended to increase computational cost but led to minimal gain in accuracy and loss. The same can be said for stacking layers. A simpler model with sufficient regularization via recurrent dropout and a dropout layer ended up being the optimal model for this task.

If one were to combine the field goal and punt predictions from our four class models with the predictions from our two class models, it is likely that that ensemble would achieve better

results, which may be an area for future research. This also may lead down the path of multi task learning, where a single architecture learns to predict different classes via model branches. An approach like this may help when attempting to predict the direction of running plays, which we were unable to do successfully with our models.

5. Conclusion

NFL play prediction has been relatively well studied since NFL play-by-play data has become widely available. By introducing sequential dependencies between plays within games, we were able to develop two models that improved on publicly available tree-based models that rely solely on the current game state. Approaches taken in this paper can likely extend to NFL front offices, where better data and resources are available to accomplish similar tasks.

REFERENCES

Carl, Sebastian and Ben Baldwin. “nflfastR: Functions to Efficiently Access NFL Play by Play Data.” https://github.com/nflverse/nflfastR, r package version 4, no. 5 (2023).

Horowitz, Maksim, Ron Yurko, and Samuel L. Ventura. "nflscrapR: Compiling the NFL play-by-play API for easy use in R." https://github.com/maksimhorowitz/nflscrapR, r package version 1, no. 0 (2017).

Jain, Rahul, Aditya Kharosekar and Keyur Muzumdar. “NFL Predictions.” https://github.com/RahulJain28/NFLPredictions, https://rahuljain28.github.io/NFLPredictions/ (2017).

Patel, Parth. “NFL TENDENCY ANALYSIS AND BASIC PLAY TYPE PREDICTION.” MSiA Student Research, January 31, 2020. https://sites.northwestern.edu/msia/2020/01/31/nfl-tendency-analysis-and-basic-play-type-prediction/.

APPENDIX

Model Information and Performance

Experiment	Primary Architecture	Targets	Training Data	Regularization	Batch Size	Sequence Length	Shuffled Sequences	Features	RNN Nodes	Train Accuracy	Train Loss	Valid Accuracy	Valid Loss	Test Accuracy	Test Loss	Time (s)
32	Bidirectional LSTM	2 (pass, run)	2020-2021*	Drop, Rec Drop, Class Wt	32	15	No	45	32	0.741	0.477	0.736	0.473	0.741	0.468	956.4
33	Bidirectional LSTM + Dense	2 (pass, run)	2020-2021*	Drop, Rec Drop, Class Wt	32	15	No	45	16	0.724	0.491	0.722	0.487	0.724	0.483	614.6
30	LSTM	4 (punt, fg, pass, run)	2020-2021*	Drop, Rec Drop	87	15	No	45	38	0.733	0.527	0.742	0.522	0.741	0.513	3016.3
31	Bidirectional LSTM	4 (punt, fg, pass, run)	2020-2021*	Drop, Rec Drop	87	15	No	45	38	0.743	0.525	0.743	0.52	0.741	0.514	2829.5
28	Bidirectional LSTM	4 (punt, fg, pass, run)	2021*	Drop, Rec Drop	87	15	No	45	64	0.727	0.55	0.732	0.543	0.711	0.589	819.1
29	Bidirectional LSTM + Dense	4 (punt, fg, pass, run)	2021*	Drop, Rec Drop	87	15	No	45	64	0.717	0.567	0.732	0.537	0.711	0.59	874.1
24	LSTM	4 (punt, fg, pass, run)	2021*	Drop, Rec Drop	54	15	Yes	45	64	0.748	0.508	0.744	0.519	0.705	0.619	945.5
27	LSTM	4 (punt, fg, pass, run)	2021*	Drop	87	15	No	45	64	0.727	0.551	0.736	0.541	0.711	0.619	535
22	Bidirectional LSTM	4 (punt, fg, pass, run)	2021	Drop, Rec Drop	64	12	Yes	54	128	0.747	0.84	0.747	0.495	0.696	0.639	731.9
6	GRU	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	100	10	No	54	32	0.712	0.8	0.718	0.767	0.716	0.76	1146.2
26	Conv 1D + Dense	4 (punt, fg, pass, run)	2021*	Drop	54	15	Yes	45	-	0.684	0.699	0.674	0.726	0.651	0.763	284.2
3	LSTM	6 (punt, fg, pass, run dir x3)	2019-2021	Drop, Rec Drop	64	15	No	57	16	0.709	0.793	0.713	0.779	0.709	0.778	8061.2
2	LSTM	6 (punt, fg, pass, run dir x3)	2019-2021	Drop	64	15	No	58	16	0.708	0.796	0.705	0.795	0.702	0.789	1035.2
25	LSTM + Dense	4 (punt, fg, pass, run)	2021*	Drop	54	15	Yes	45	8	0.722	0.572	0.722	0.561	0.675	0.812	572.4
1	LSTM	6 (punt, fg, pass, run dir x3)	2019	None	64	15	No	58	16	0.708	0.809	0.936	0.853	0.688	0.847	156.5
4	GRU	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	64	20	No	57	32	0.68	0.893	0.679	0.867	0.684	0.854	1871.9
19	Bidirectional LSTM	4 (punt, fg, pass, run)	2021	Drop, Class Wt	64	21	Yes	54	24	0.701	0.605	0.7	0.605	0.7	0.883	750.7
23	Stacked Bidirectional LSTM	4 (punt, fg, pass, run)	2021	Drop, Rec Drop	54	15	Yes	54	124, 68	0.744	0.508	0.744	0.516	0.629	1.14	1446.17
20	Bidirectional LSTM	4 (punt, fg, pass, run)	2021	Drop	64	12	Yes	54	16	0.72	0.544	0.731	0.532	0.629	1.403	641.6
18	Bidirectional LSTM	4 (punt, fg, pass, run)	2021	Drop, Class Wt	64	21	Yes	54	32	0.622	0.646	0.648	0.718	0.539	1.971	898.4
15	Bidirectional GRU	6 (punt, fg, pass, run dir x3)	2021	Drop, Class Wt	68	21	Yes	54	16	0.454	1.17	0.48	1.26	0.414	2.59	353.3
17	Stacked Bidirectional LSTM	6 (punt, fg, pass, run dir x3)	2021	Drop, Rec Drop, Class Wt	68	21	Yes	54	16, 16	0.642	0.51	0.643	1.012	0.545	2.844	927.5
5	GRU	6 (punt, fg, pass, run dir x3)	2020-2021	PCA, Drop	128	20	No	21	32	0.287	-	0.286	-	-	-	-
7	GRU	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	100	10	No	54	64	0.699	0.912	0.709	0.859	-	-	-
8	Bidirectional LSTM	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	100	10	No	54	64	0.721	0.764	0.721	0.76	-	-	-
9	Stacked Bidirectional LSTM	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	100	10	No	54	32, 32	0.664	1.27	0.68	1.103	-	-	-
10	Stacked Bidirectional LSTM + Dense	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	100	10	No	54	128, 64	0.71	0.8	0.715	0.766	-	-	-
11	Stacked Bidirectional LSTM	6 (punt, fg, pass, run dir x3)	2020-2021	Drop, Rec Drop	100	10	No	54	254, 128	0.706	0.96	0.718	0.877	-	-	-
12	Bidirectional LSTM	6 (punt, fg, pass, run dir x3)	2020-2021	Drop, Rec Drop	100	10	No	54	64	0.681	1.14	0.687	1.056	-	-	-
13	Stacked Bidirectional LSTM + Dense	6 (punt, fg, pass, run dir x3)	2020-2021	Drop, Rec Drop	100	10	No	54	4	0.699	0.815	0.705	0.803	-	-	-
14	GRU	6 (punt, fg, pass, run dir x3)	2020-2021	Drop	128	5	Yes	54	32	0.721	0.772	0.727	0.743	-	-	-
16	Stacked Bidirectional LSTM	6 (punt, fg, pass, run dir x3)	2021	Drop, Class Wt	68	21	Yes	54	16, 16	0.501	0.824	0.463	1.264	-	-	-
21	Bidirectional LSTM	4 (punt, fg, pass, run)	2021	Drop, Rec Drop	64	12	Yes	54	32	0.719	0.588	0.719	0.579	-	-	-

* updated set of features

Figure 1: Class imbalance with 6 target classes

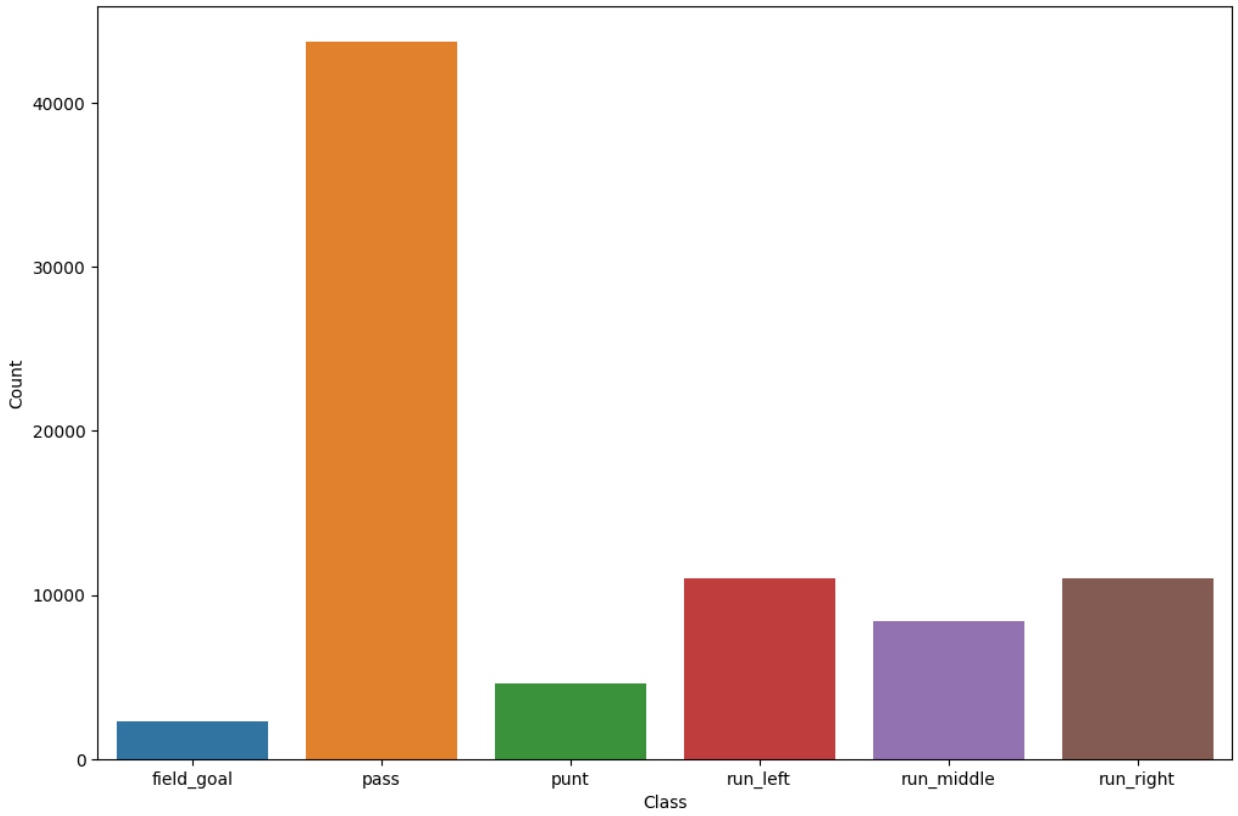

Figure 2: Class imbalance with 4 target classes

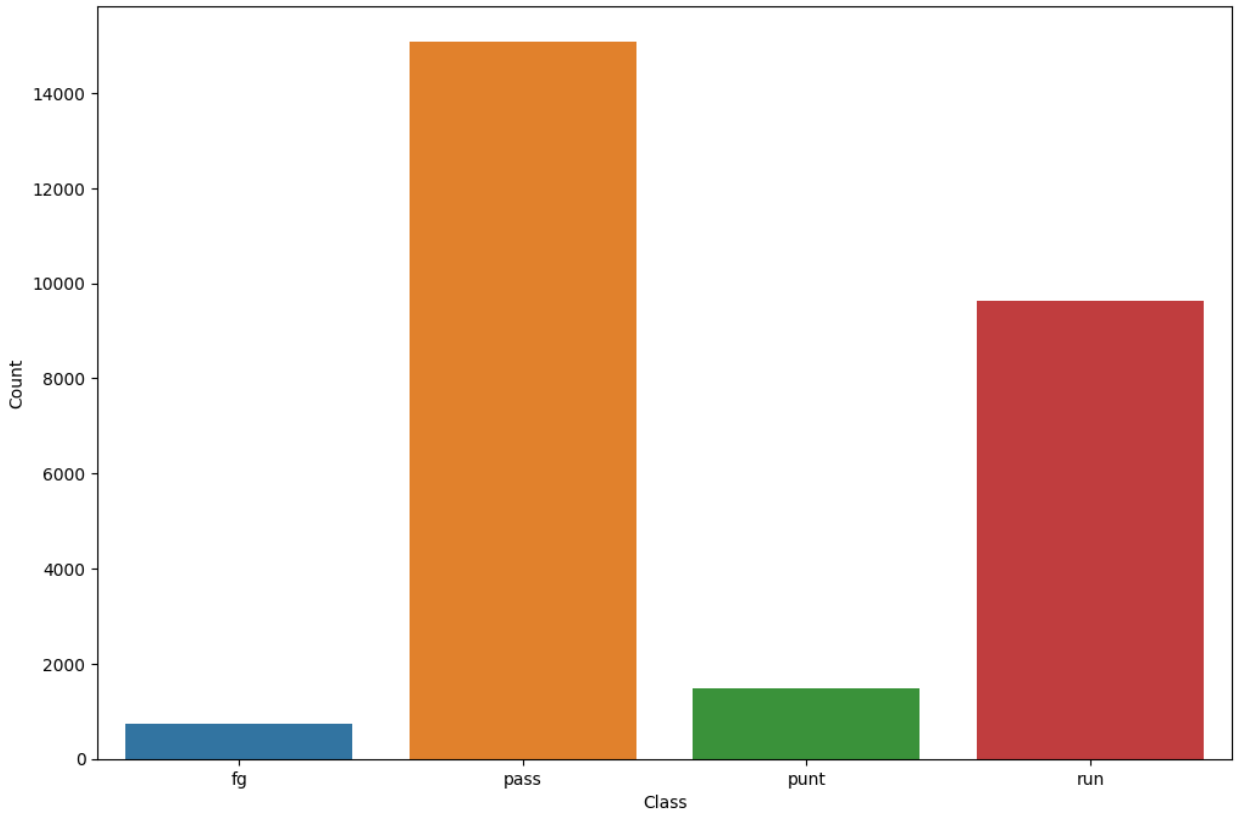

Figure 3: Class imbalance with 2 target classes

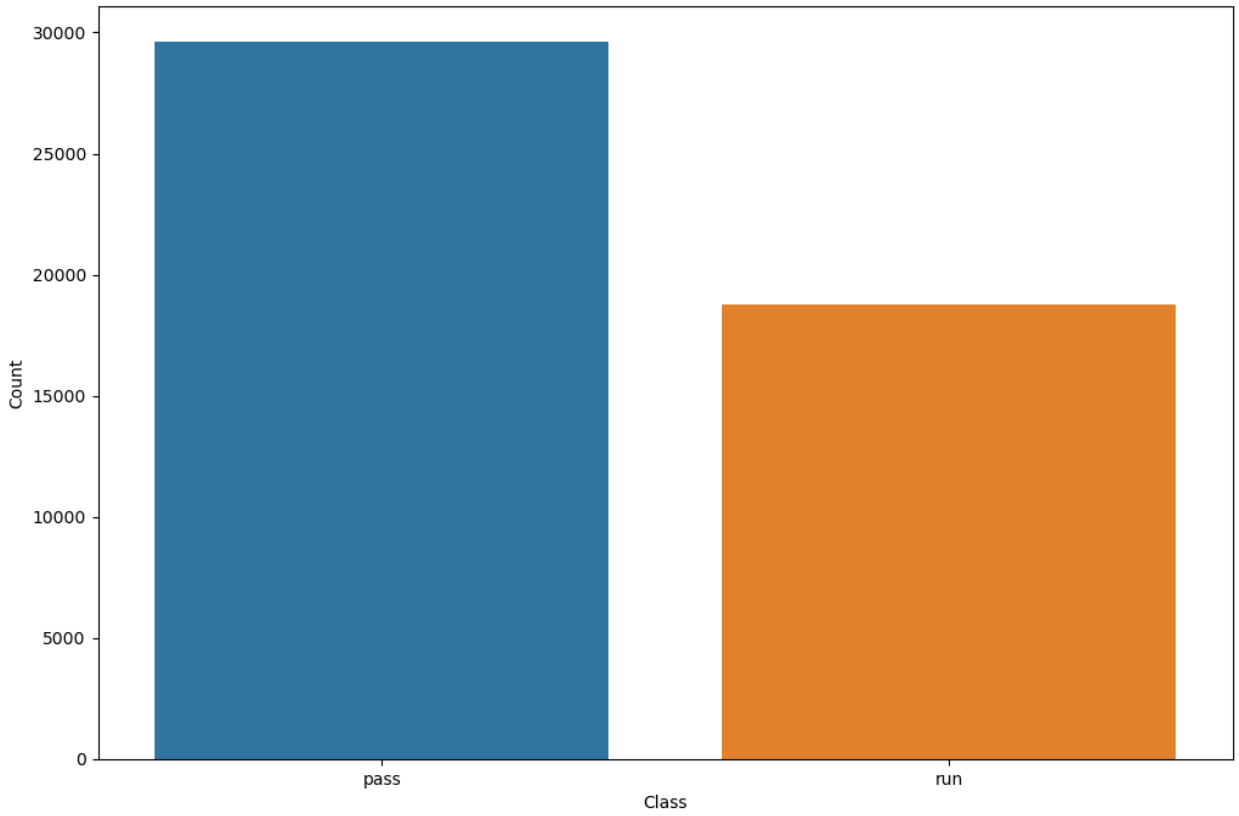

Figure 4: Training and Validation Curves (Exp 01)

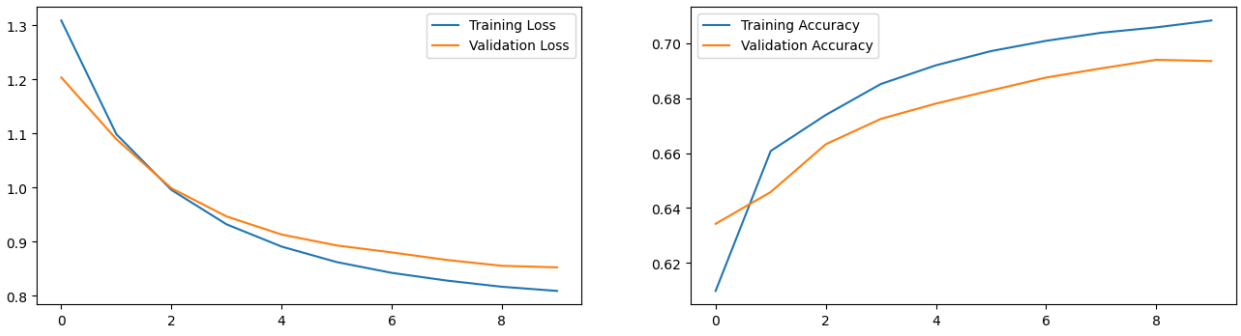

Figure 5: Confusion Matrix (EXP 01)

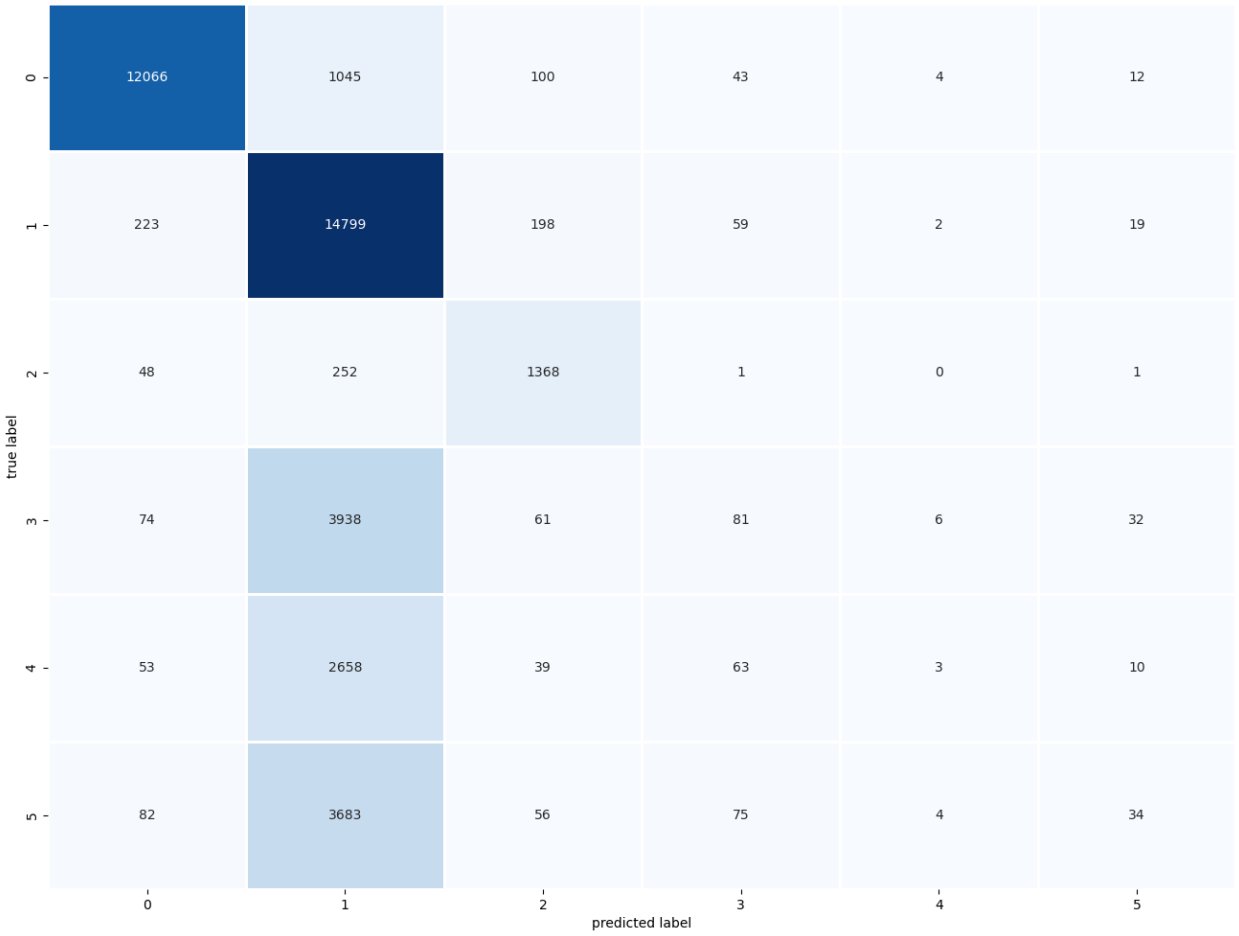

Figure 6: RF Feature Importance (Exp 01)

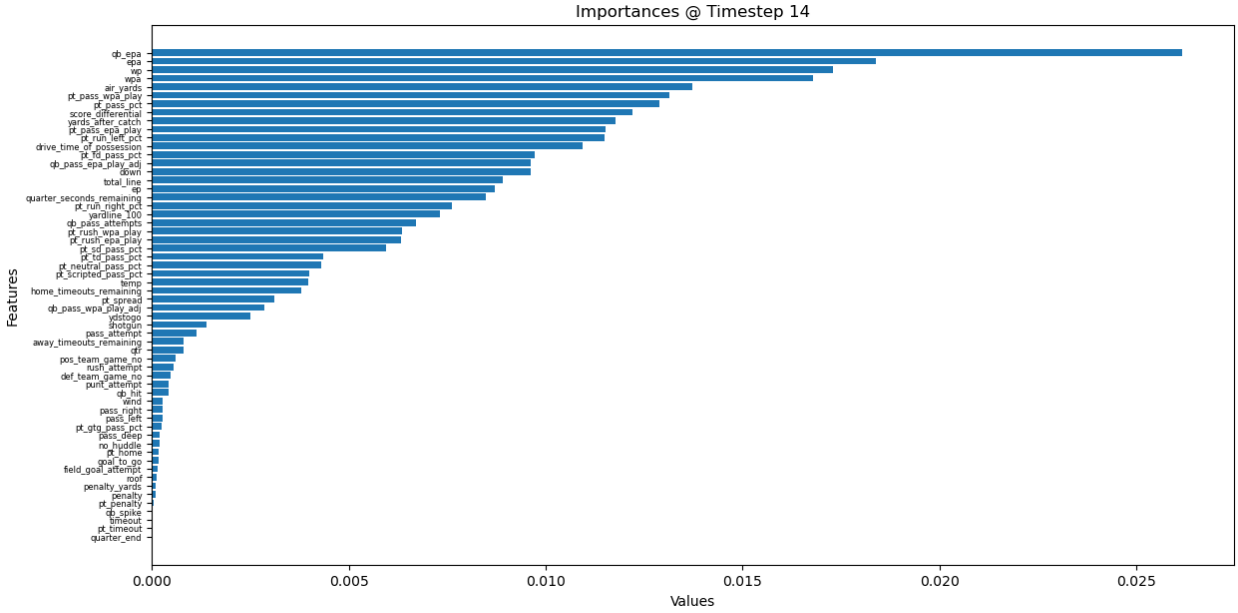

Figure 7: Confusion Matrix (Exp 03 LSTM)

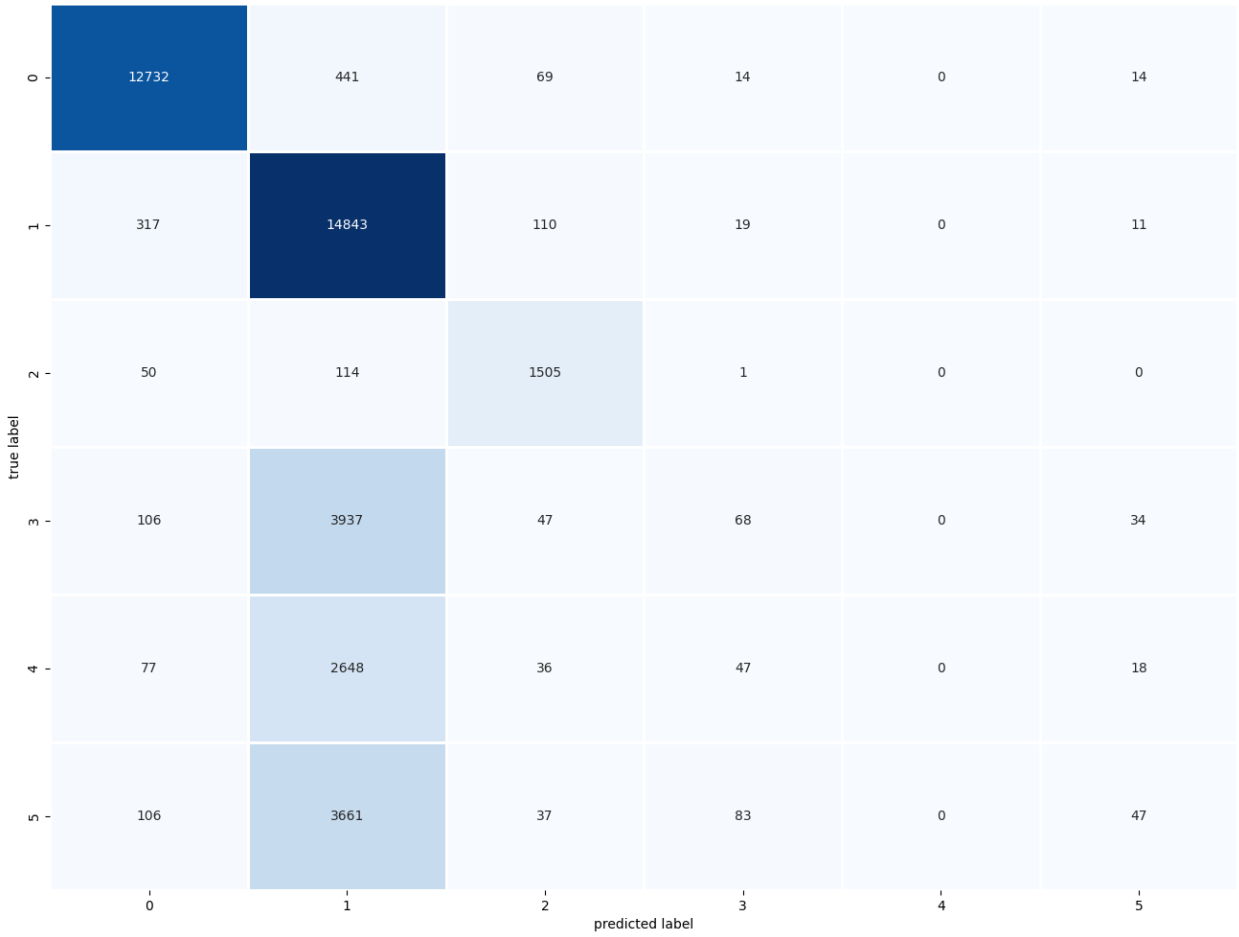

Figure 8: Confusion Matrix - LSTM with Class Weights (EXP 16)

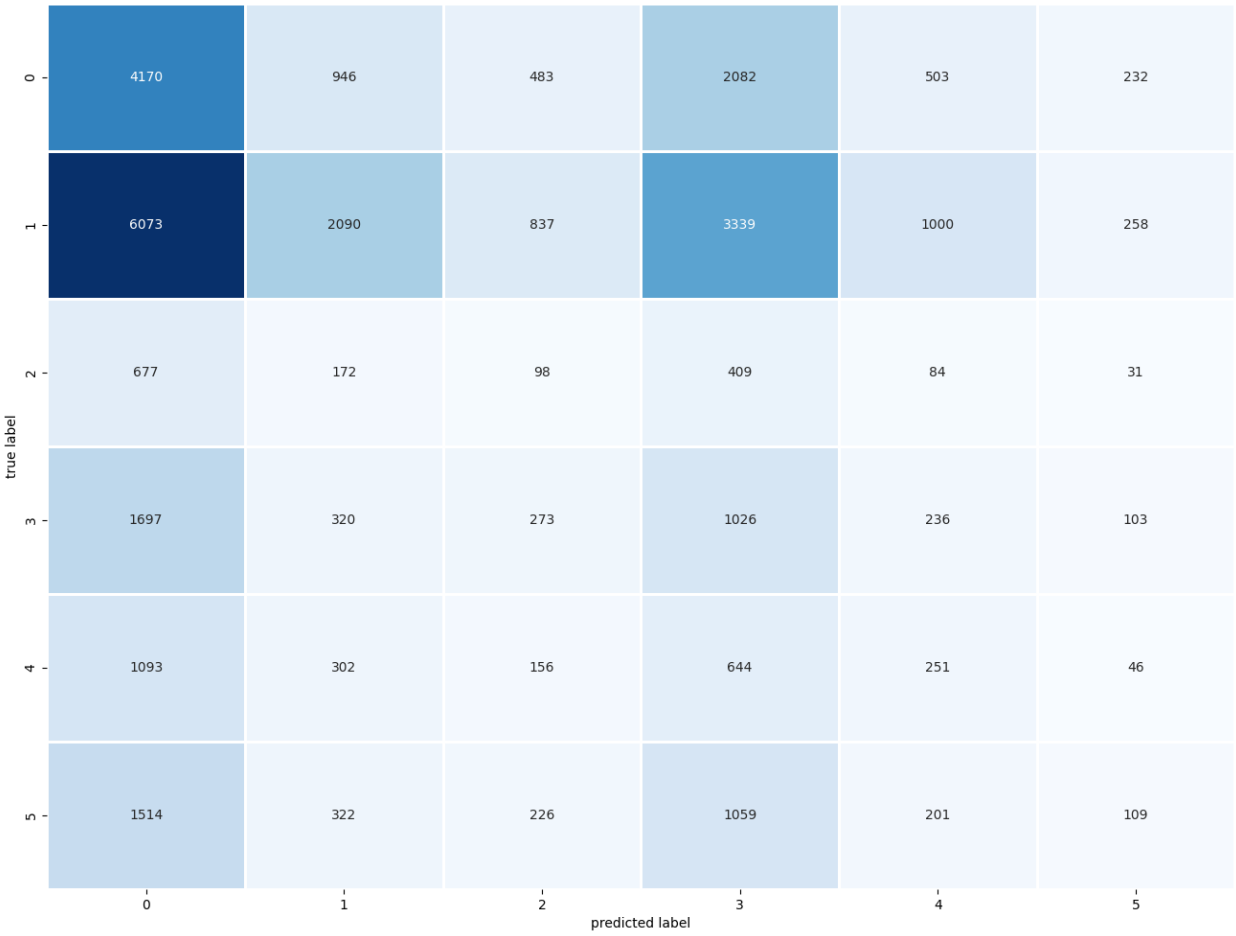

Figure 9: Training and Validation Curves (EXP 24)

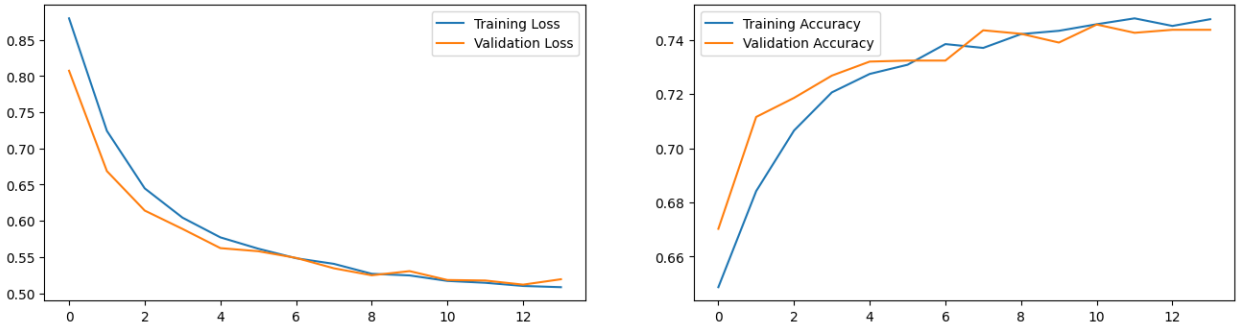

Figure 10: Confusion Matrix (EXP 24)

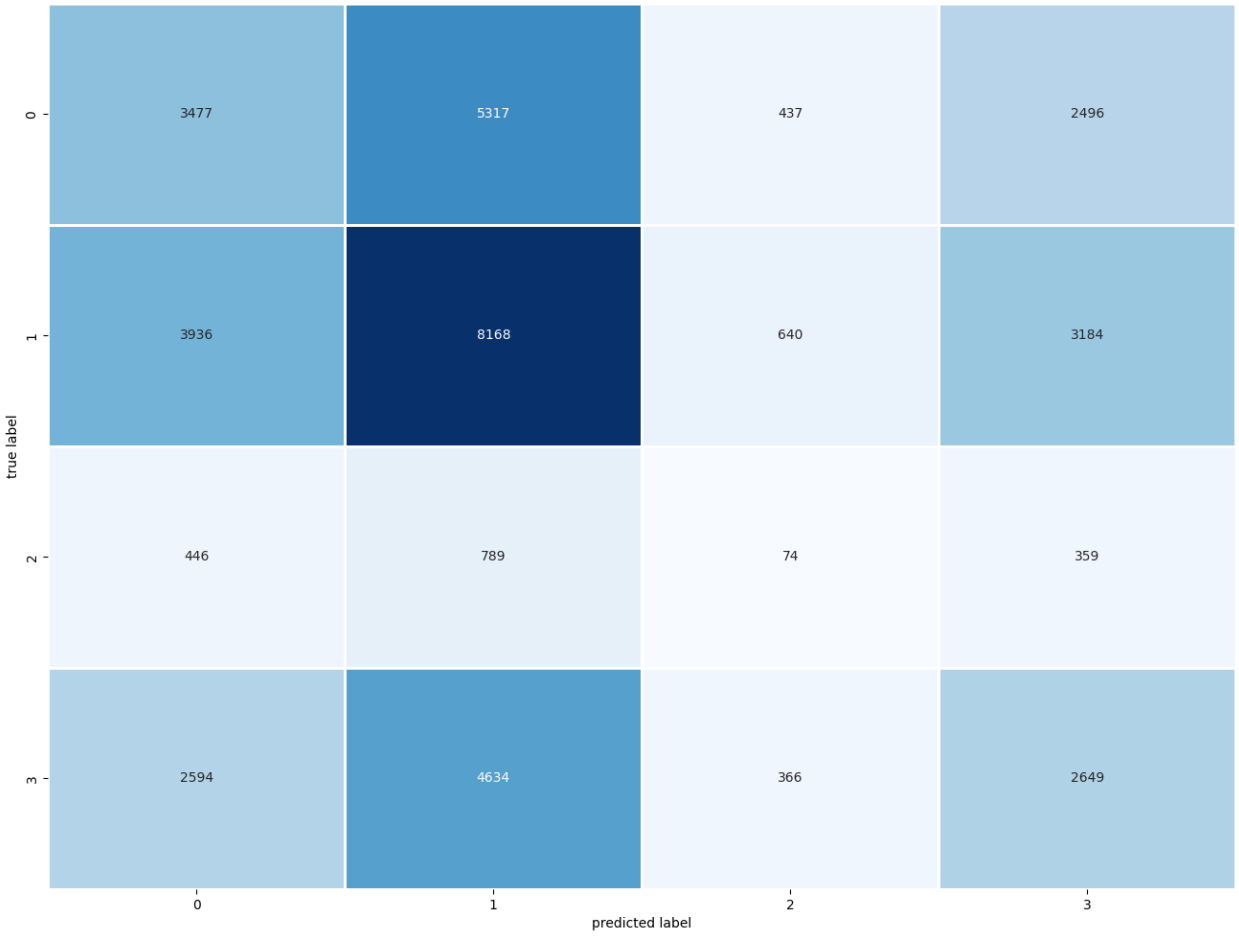

Figure 11: Training and Validation Curves (EXP 28)

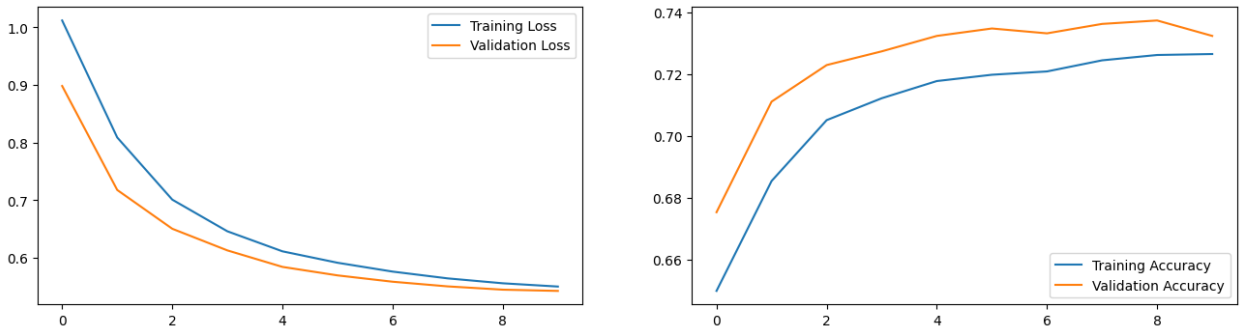

Figure 12: Confusion Matrix (EXP 28)

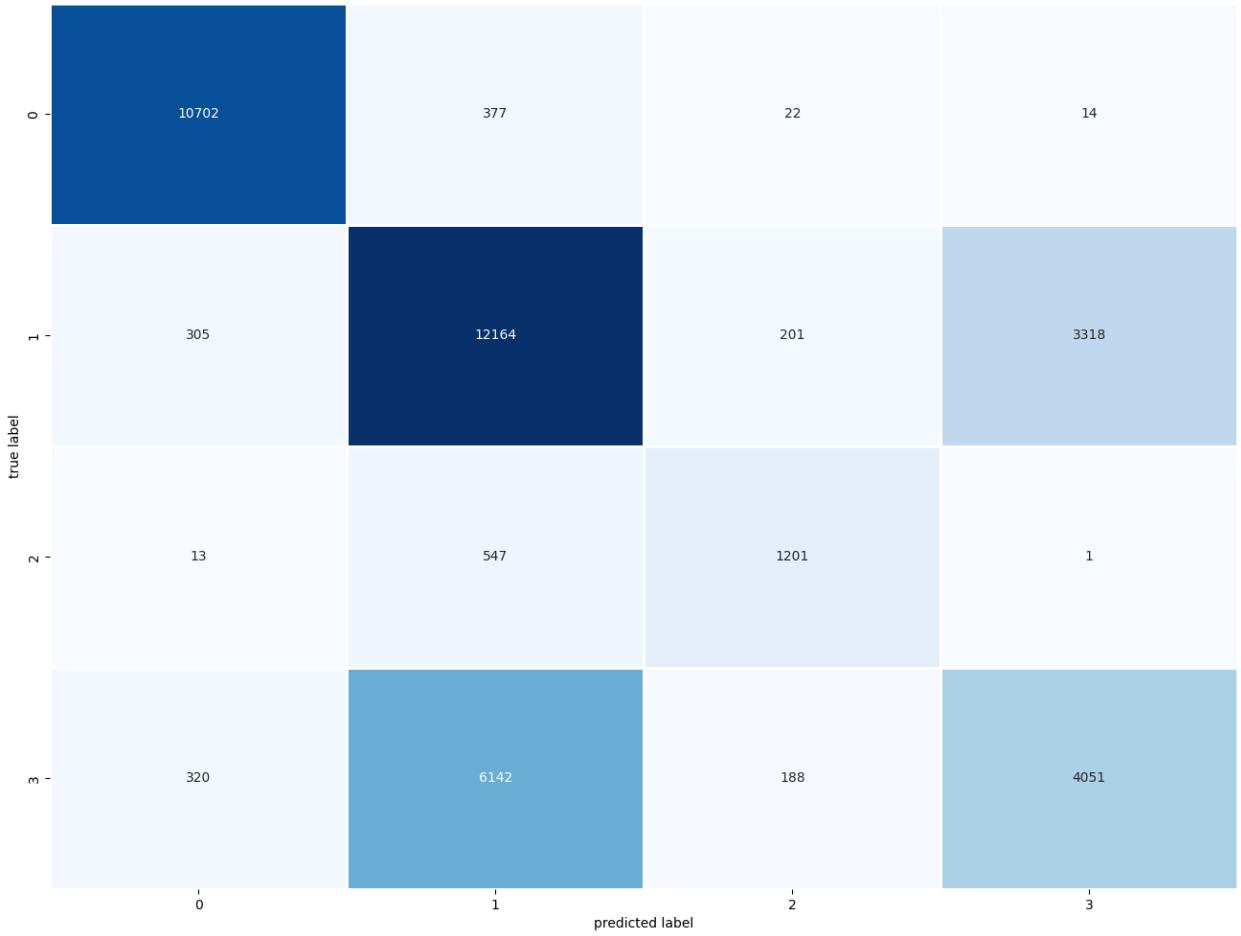

Figure 13: Classification Report (EXP 28)

Classification Report

	precision	recall	f1-score	support
0	0.94	0.96	0.95	11115
1	0.63	0.76	0.69	15988
2	0.75	0.68	0.71	1762
3	0.55	0.38	0.45	10701
accuracy			0.71	39566
macro avg	0.72	0.70	0.70	39566
weighted avg	0.70	0.71	0.70	39566

Figure 14: Confusion Matrix (EXP 31)

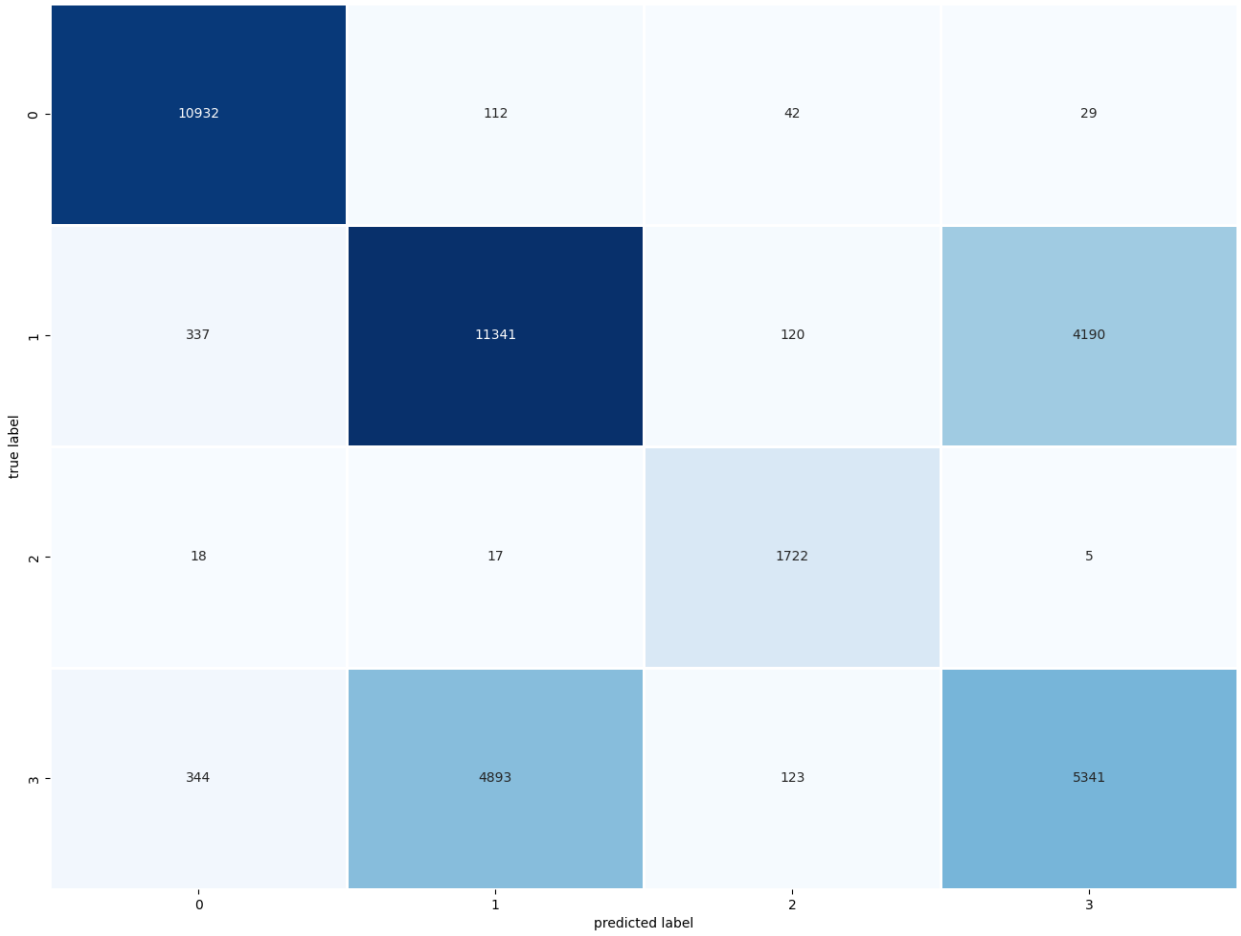

Figure 15: Classification Report (EXP 31)

Classification Report

	precision	recall	f1-score	support
0	0.94	0.98	0.96	11115
1	0.69	0.71	0.70	15988
2	0.86	0.98	0.91	1762
3	0.56	0.50	0.53	10701
accuracy			0.74	39566
macro avg	0.76	0.79	0.78	39566
weighted avg	0.73	0.74	0.74	39566

Accuracy Score: 0.7414446747207198
Root Mean Square Error: 1.013518791486649

Figure 16: Classification Report (Exp 33)

```
Classification Report
              precision    recall  f1-score   support

           0       0.87      0.75      0.80     23930
           1       0.54      0.72      0.61      9674

    accuracy                           0.74     33604
   macro avg       0.70      0.73      0.71     33604
weighted avg       0.77      0.74      0.75     33604

Accuracy Score: 0.7405368408522794
Root Mean Square Error: 0.5093752635805163
```