CS 7641 Final Report - ML Poker Bot

Introduction and Background:

Poker is a challenging environment for decision-making due to its inherent uncertainty. Players must rely on incomplete information, knowing only their cards and the community cards, and calculate their odds of winning based on this. The game's probabilistic nature and delicate balance of risk and reward have made it a popular application for machine learning (ML) [1]. Recent advancements like DeepStack [2] and Libratus [3], have showcased the potential of ML in creating poker bots that excel against human opponents. Building on this research, we will create a fully autonomous poker bot that analyzes positions, performs standard Texas Hold'em actions, and implements bluffing to maximize earnings and unpredictability.

We used the Pluribus Poker Hand Dataset [5] to group poker hands through clustering (unsupervised learning) and to build models that classify hand strengths and predict actions (supervised learning). We will refine the bot's decision-making using NeuronPoker [4], an environment that uses reinforcement learning to train neural networks to play poker. Users will be able to play with the bot in real time through PyPokerEngine [6].

Problem Definition:

Poker's incomplete information creates uncertainty, strategic complexity, and psychological stress, making optimal decisions challenging. Humans struggle to consistently evaluate hand strength and predict the best actions in dynamic game environments because of this. However, a bot that leverages ML can analyze datasets to adapt its strategy, use probabilistic modeling to make real-time decisions, and scrutinize opponent behavior for insights. Developing a poker-playing bot will optimize in-game decisions, learn from outcomes, and adjust strategies to outperform humans.

Methods:

Data Preprocessing:

Unprocessed Data Chart

	Game_ID	Session_ID	Player_Name	Position	dealt_cards_S1	dealt_cards_C1	dealt_cards_S2
Eddie	30,003	30	Eddie	1	11	1	13
Bill	30,003	30	Bill	2	11	2	10
Pluribus	30,003	30	Pluribus	3	3	1	14
MrWhite	30,003	30	MrWhite	4	4	4	10
Gogo	30,003	30	Gogo	5	5	1	14
Budd	30,003	30	Budd	6	9	3	9

We utilize data from poker games [7], formatted in the PokerStars hand history—a semi-structured text format—to extract detailed information about poker hands, including player actions, cards, and outcomes. Our Python script preprocesses this data into a machine learning-ready tabular format. Each hand is transformed into numerical features: cards are encoded by suit (1-4) and rank (1-13); player actions are categorized (0 = none, 1 = fold, 2 = call, 3 = raise, 4 = check); and player positions are specified (0 = small blind, 1 = big blind, 2 = other, 3 = button). This encoding preserves poker-specific patterns, enabling a consistent representation of actions and outcomes across betting rounds (preflop to river) in sequential numerical features.

Processed Data

	Game_ID	Session_ID	Player_Name	total_players	Position	dealt_cards_S1	dealt_cards_C1	dealt
0	87,000	87	Bill	6	2	4	4	
1	87,000	87	Eddie	6	1	4	5	
2	87,000	87	MrBlue	6	2	4	1	
3	87,000	87	MrOrange	6	2	4	13	
4	87,000	87	MrPink	6	0	3	11	

In order to get our data in a suitable format to analyze using `K-Means`, we organized our dataset into categorical and numerical features and then employed two key data preprocessing methods. First. we used `StandardScaler` to standardize features like bet sizes and pot odds to ensure consistent scales. Then, we used `OneHotEncoder` to convert categorical features like card suits and player positions into numerical formats to feed into machine learning algorithms.

```
# kmeans clustering preprocessing code
def preprocess_data(self):
        numerical_features = [
            "Preflop_Amount",
            "Flop_Amount",
            "Turn_Amount",
            "River_Amount",
            "Pot_Size",
        ]
        categorical_features = ["Position"]

        numerical_transformer = StandardScaler()
        categorical_transformer = OneHotEncoder(handle_unknown="ignore")

        self.preprocessor = ColumnTransformer(
            transformers=[
                ("num", numerical_transformer, numerical_features),
                ("cat", categorical_transformer, categorical_features),
            ]
        )

        self.features = self.preprocessor.fit_transform(self.data)
```

Our preprocessing effort to get our data into a format suitable for our supervised learning methods (Random Forest and XGBoost) was a little more intensive. We first loaded the data from the excel file using simple python commands. We then used `OneHotEncoder` to convert any categorial features into numerical for easier processing, similar to K-Means. We then applied feature engineering to create new features such as `pot_size_ratio`, `hand_card1_rank` and `suited` which are important for feeding into our `Random Forest` classifier. For our data, we decided to use a binary model which would result in a 1 if the card was present and a 0 if the card was not. This simplification was incredibly important to being able to analyze these features in our algorithm. Lastly, we used `StandardScaler` to standardize features like bet sizes and pot odds to ensure consistent scales for our numerical features. We also applied label encoding in our `prepare_target` method to create labels for our `fold`, `check/call`, and `bet/raise` actions. All of this can be seen in the code below. The full model can be found in the `supervised` folder.

```
# random forest preprocessing code
def load_and_prepare_data(self):
        self.data = pd.read_excel(self.data_path)
        self.prepare_features()
        self.prepare_target()

    def prepare_features(self):
```

```python
        self.features = pd.DataFrame()

        position_dummies = pd.get_dummies(self.data["Position"], prefix="pos")
        self.features = pd.concat([self.features, position_dummies], axis=1)
        self.features["pot_size_ratio"] = (
            self.data["Pot_Size"] / self.data["total_players"]
        )

        self.features["hand_card1_rank"] = self.data["dealt_cards_C1"]
        self.features["hand_card2_rank"] = self.data["dealt_cards_C2"]
        self.features["suited"] = (
            self.data["dealt_cards_S1"] == self.data["dealt_cards_S2"]
        ).astype(int)

        for street in ["flop", "turn", "river"]:
            self.features[f"{street}_present"] = (
                self.data[f"{street}_C1"] != 0
            ).astype(int)

        self.features["prev_preflop_amount"] = (
            self.data["Preflop_Amount"].shift(1).fillna(0)
        )
        self.features["prev_flop_amount"] = self.data["Flop_Amount"].shift(1).fill
        self.features["prev_turn_amount"] = self.data["Turn_Amount"].shift(1).fill

        numerical_columns = [
            "pot_size_ratio",
            "hand_card1_rank",
            "hand_card2_rank",
            "prev_preflop_amount",
            "prev_flop_amount",
            "prev_turn_amount",
        ]
        self.features[numerical_columns] = self.scaler.fit_transform(
            self.features[numerical_columns]
        )

    def prepare_target(self):
        def get_action(row):
            if row["Preflop_Action"] == 0:
                return 0
            elif row["Preflop_Amount"] == 0:
                return 1
            return 2
        self.target = self.data.apply(get_action, axis=1)
```

Unsupervised Learning Method

K-Means Clustering (`scikit-learn`) was used to group hands based on similarity, providing insight into hand types for strategy decisions. We used this to create a new attribute that evaluates the strength for each poker hand we receive. Eventually, we will feed this into our supervised learning model to predict the bot's actions. You can view our K-Means model in the github repository under the `kmeans` folder.

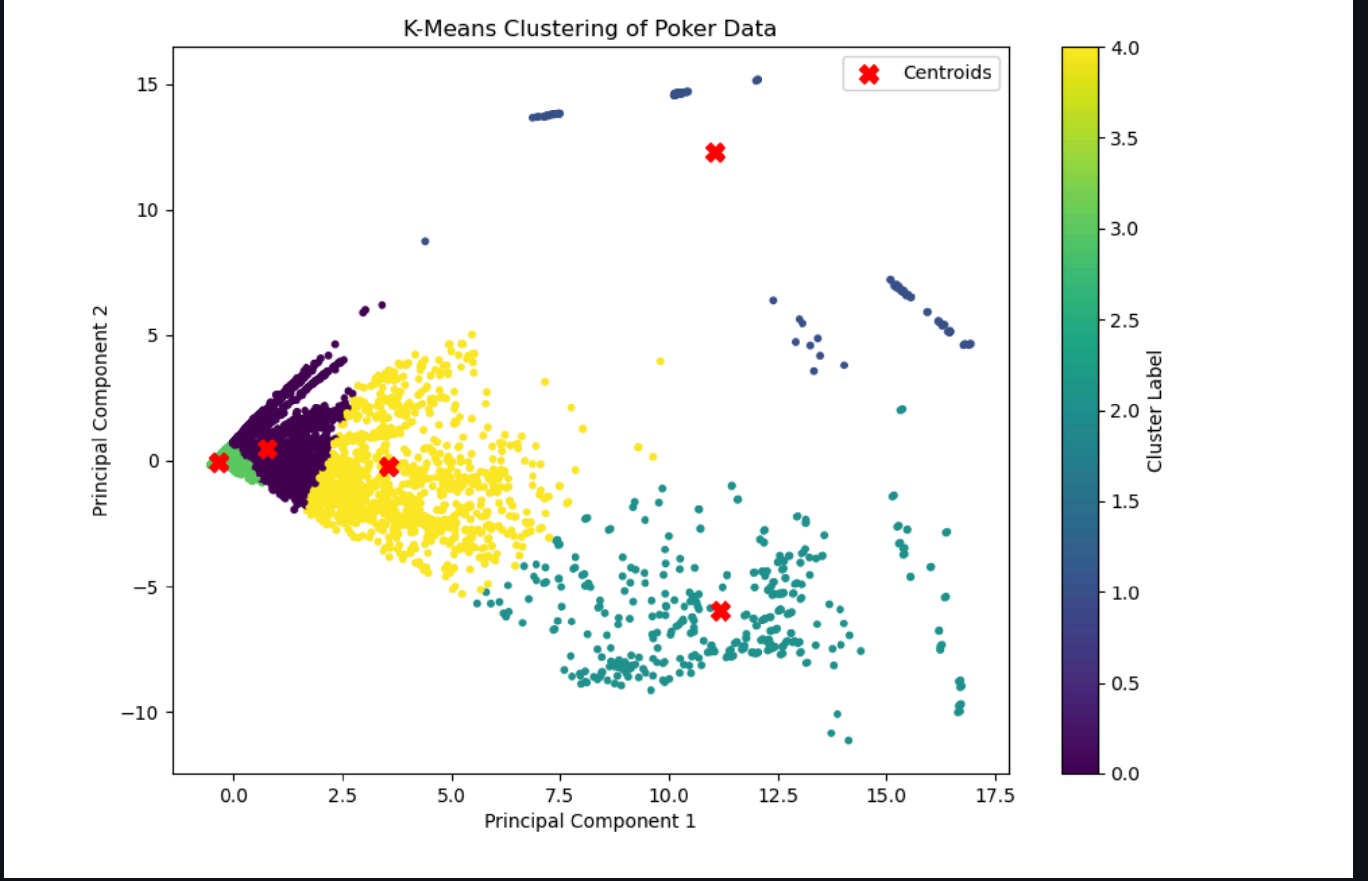
Supervised Learning Methods

RandomForestClassifier (`scikit-learn`) was used for supervised learning to predict the bot's actions based on the hand and game context. Currently, we feed in the pre-flop actions of the players before the flop is revealed, along with the card at the turn to predict determine if the bot should call/raise or fold. Notably, we do not yet consider bet size, pot odds, or any of the other factors that go into making a decision, and this is part of our next steps to further refine the model.You can view our RandomForest model in the github repository under the `supervised` folder.

We also used the **XGBClassifier** (`xgboost`) as a supervised learning approach to predict the bot's actions based on the hand and game context. This model takes in as input the pre-flop actions of the player and the card at the turn to determine what decision the bot should make, whether call, raise, or fold. We chose XGBoost specifically because of its use of gradient boosting techniques that might give it a performance advantage over our KMeans and Random Forest Classifiers. The model also includes visualizations for confusion matrices and feature importancesWe still have to incorporate bet size, pot odds, and other factors to have our model make the best possible decision, but the entire functionality for the model, including testing and visualizationsis discussed in the results and discussion below. You can view our xgboost model in the github repository under the `supervised` folder.

Results and Discussion:

Visualization and Quantitative Metrics: K-Means



```
Cluster 0: Center = [0.75766589 0.46431815], Size = 6596
Cluster 1: Center = [11.05555648 12.26432369], Size = 183
Cluster 2: Center = [11.17496392 -5.97153325], Size = 379
Cluster 3: Center = [-0.35773694 -0.05137984], Size = 50891
Cluster 4: Center = [ 3.54033734 -0.22386638], Size = 1951
```

```
Performing clustering analysis with K-Fold cross-validation...
Fold 1: Silhouette Score = 0.7289
Fold 2: Silhouette Score = 0.7261
Fold 3: Silhouette Score = 0.7241
Fold 4: Silhouette Score = 0.7290
Fold 5: Silhouette Score = 0.7199
Average Silhouette Score across folds: 0.7256
```

From the visualization we can see that our data points are grouped based on features after the PCA transformation. We can see that Clusters 0 and 3 are centered near the origin, which indicates they represent low-intensity scenarios such as positions common across players or low bet amounts. Clusters 1 and 2 on the other hand have less data values than cluster 0 and 3, indicating more intense and less frequent positions, and they have larger coordinate values to signify larger bets that may shake up the game. Cluster 4 seems to be neither low intensity nor high intensity positions, and represents mid game scenarios.

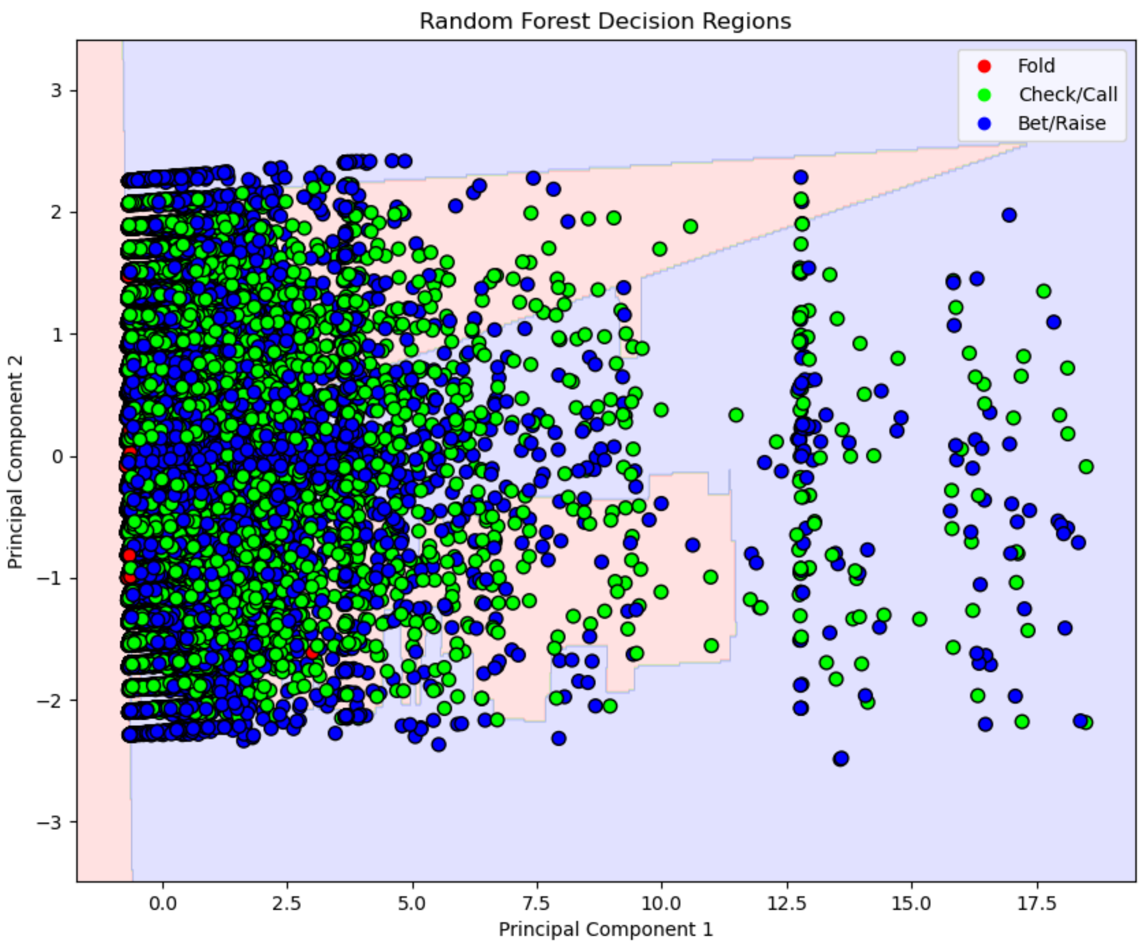
We can also clearly see that most of the data points fall under Cluster 3. This aligns with our suggestion that this cluster represents low intensity, common moves and are more conservative, as this is what most poker actions typically are. Clusters 1 and 2 have less data but higher coordinate values, indicating that these more risky moves are less played and have significant deviation from normal moves.

We also computed Silhouette scores using K-fold cross validation, and our average was 0.7256. Silhouette scores range from -1 to 1, so a value of 0.7256 indicates strong separability and well-defined clusters. It also indicates that our K-means algorithm did well to group these Poker actions into similar clusters that are very different from each other, which is the key to the most effective grouping. Our model performed well because of our preprocessing and the similarity of actions in poker games. We're going to have a lot of conservative actions (denoted by closer to the origin) but also a fair amount of risky plays (denoted by larger x and y coordinate values). As a result, we had a lot of data points with similar values, and we were able to easily form clusters that were not only similar within the cluster but also different from the other clusters.

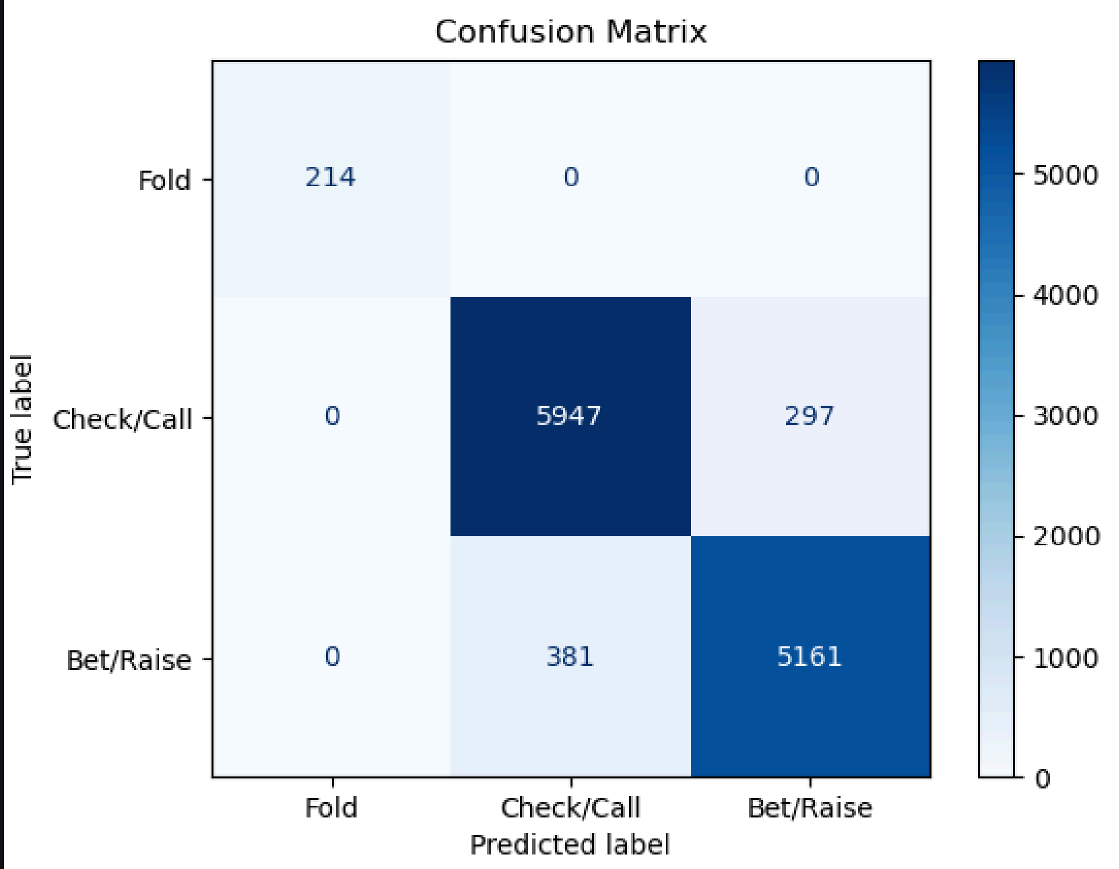
We can see how we generated the K-Fold Validation Clustering metrics here:

```python
def cross_validate_clustering(self, n_splits=5):
        kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
        silhouette_scores = []

        for fold, (train_index, test_index) in enumerate(kf.split(self.features)):
            X_train, X_test = self.features[train_index], self.features[test_index]

            kmeans = KMeans(n_clusters=self.n_clusters, random_state=0, n_init=10)
            kmeans.fit(X_train)

            labels = kmeans.predict(X_test)
            score = silhouette_score(X_test, labels)
            silhouette_scores.append(score)
            print(f"Fold {fold + 1}: Silhouette Score = {score:.4f}")

        average_score = np.mean(silhouette_scores)
        print(f"Average Silhouette Score across folds: {average_score:.4f}")
```

Visualization and Quantitative Metrics: Random Forest Classification



```
Model Performance:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       214
           1       0.94      0.95      0.95      6244
           2       0.95      0.93      0.94      5542

    accuracy                           0.94     12000
   macro avg       0.96      0.96      0.96     12000
weighted avg       0.94      0.94      0.94     12000
```



Analysis of Random Forest Model

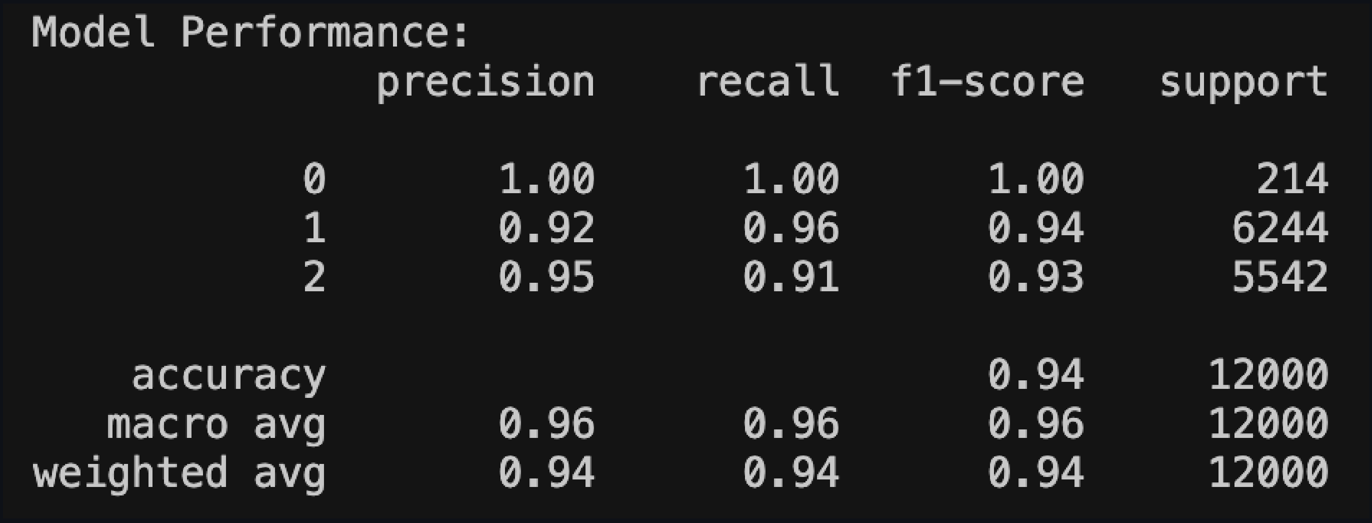
Based on the model performance metrics, we can see that the model has an accuracy of 94%, and the precision, recall, and F1 score for each of the classes (`Fold` , `Check/Call` , `Bet/Raise`). The model performs particularly excellently on the `Fold` class as it had no misclassifications. For the `Check/Call` and `Bet/Raise` classes, the precision scores are 0.94 and 0.95, respectively, and their recall scores are 0.95 and 0.93, respectively. This emphasizes the ability for the model to effectively understand the difference between these actions (in terms of feature relations). While it isn't 1.00 like the fold class, the overall precision and recall scores for `Check/Call` and `Bet/Raise` classes are extremely high and balanced, indicating a very well-performing model.

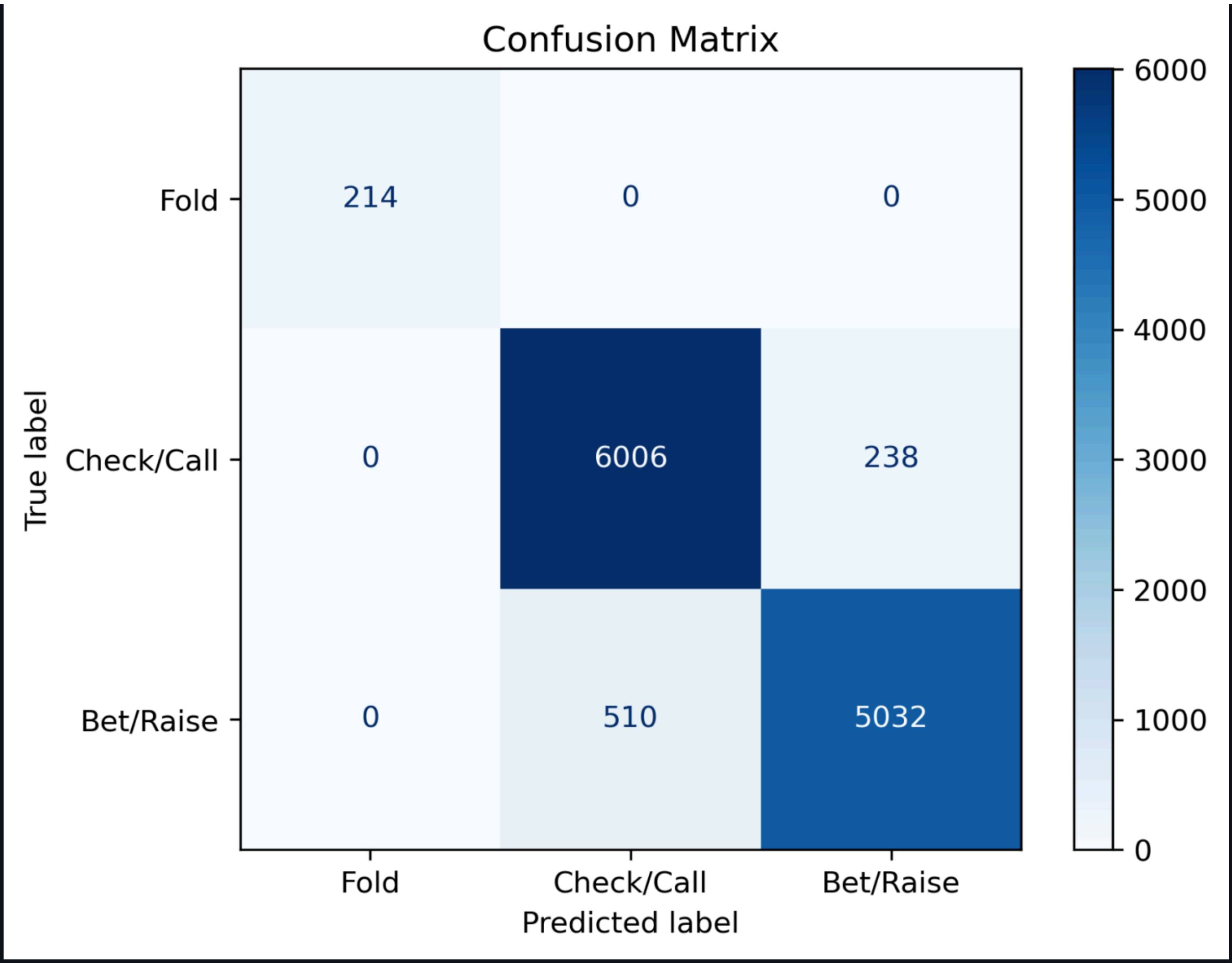
From the confusion matrix, we can see that our model performed particularly well when predicting when to `Check/Call` and `Bet/Raise` (heavy prominence on the diagonal). It predicted folding perfectly as well, with all **214** samples identified correctly. There were some misclassifications however, as **297** samples of `Check/Call` were misclassified as `Bet/Raise` and, on the other hand, **381** cases of `Bet/Raise` were misclassified as `Check/Call` . As such, roughly **5.65%** of the dataset was misclassified, which is a sizable amount. This error likely is a result of the complicated nature behind poker decisions and the number of features that are involved. Additionally, the feature distributions for certain actions likely overlap, making it difficult for the model to distinguish between the two actions. However, the model does still correctly identify a large majority of the actions, and our feature engineering to emphasize pot size ratio, card hand rankings, etc. played a part in that.

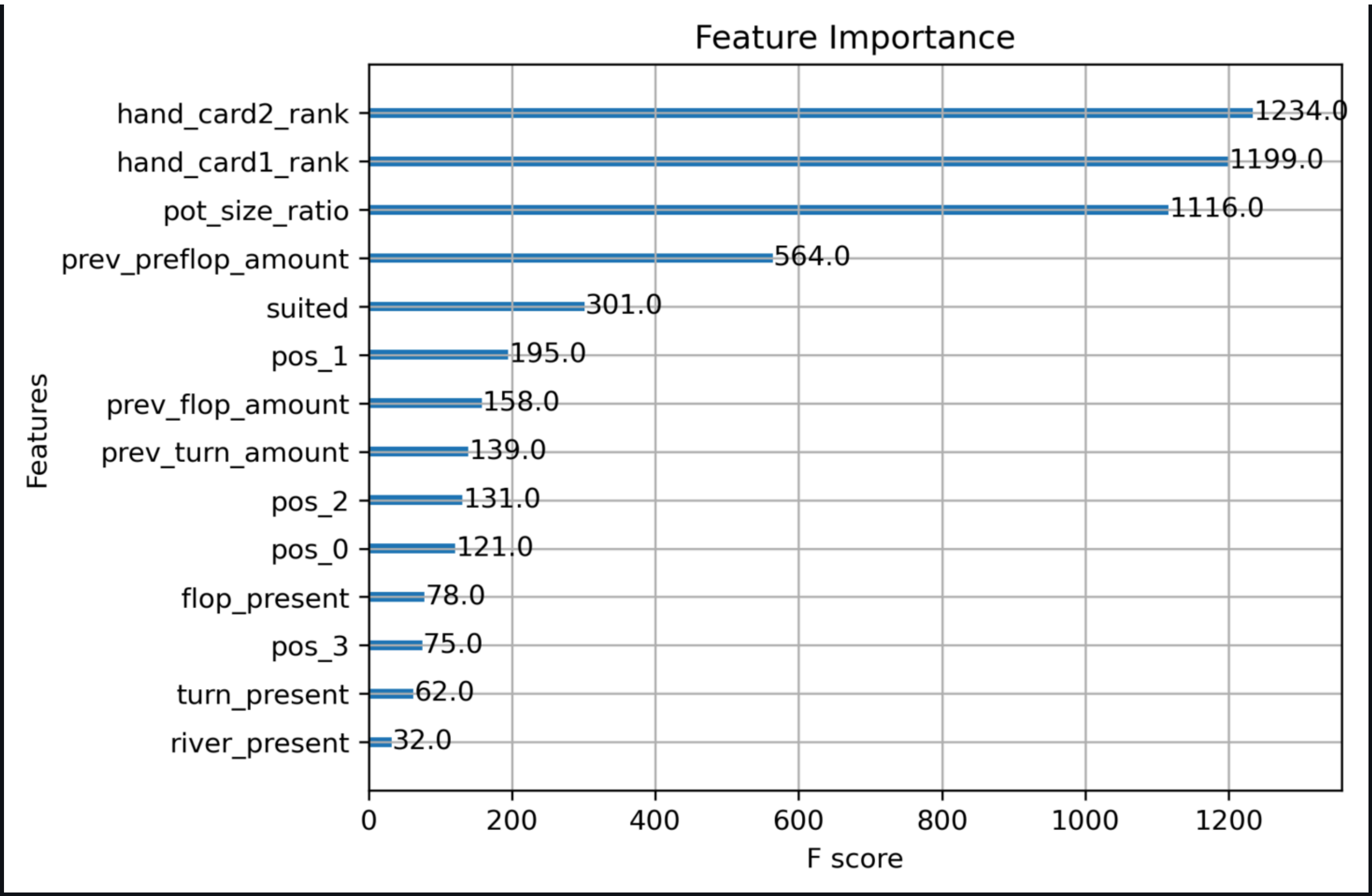
In terms of weaknesses, we might have to oversample from the `Fold` class as it has a lot less samples than any of the other classes. We also could add more features that are unique to each action or better tune our hyperparameters to prevent the misclassification that is currently occuring. These would help with the recall and precision metrics that we discussed above as well as these misclassifications were the reason for slightly lower scores.

Ultimately, the random forest classifier performs strongly on predicting the users next action based on the hand and game context. In addition to the weaknesses, there are a few key limitations of the model that we will need to address in the future. First, we have not yet considered the relative strength of each player in order to determine if their actions are worth trading on. Second, we have not yet considered factors such as bet sizing, pot odds, and other factors that go into making a decision in poker. We have a model with a ternary output to predict the next action (fold / check / raise) but we have not yet considered the nuances of the game. We used 100 estimators as that was best on our quick, empirical tests, but more experimentation is required.

Visualization and Quantitative Metrics: XGBoost Model


```
Model Performance:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       214
           1       0.92      0.96      0.94      6244
           2       0.95      0.91      0.93      5542

    accuracy                           0.94     12000
   macro avg       0.96      0.96      0.96     12000
weighted avg       0.94      0.94      0.94     12000
```


Confusion Matrix
True label
Fold
Check/Call
Bet/Raise
Predicted label
214
0
0
0
6006
238
0
510
5032
6000
5000
4000
3000
2000
1000
0



Analysis of the XGBoost Model

Based on the model performance metrics, we can see that the model has an accuracy of 94%, and the precision, recall, and F1 score for each of the classes (`Fold` , `Check/Call` , `Bet/Raise`). The model performs particularly excellently on the `Fold` class as it had no misclassifications. For the `Check/Call` and `Bet/Raise` classes, the precision scores are 0.92 and 0.95, respectively, and their recall scores are 0.96 and 0.91, respectively. This emphasizes the ability for the model to effectively understand the difference between these actions (in terms of feature relations). While it isn't 1.00 like the fold class, the overall F1 scores of 0.94 for `Check/Call` and 0.93 for `Bet/Raise` indicate strong performance for the model.

How is this information compiled From the feature importance chart, we can see that the three most important features, by a significant margin, are `hand_card2_rank` , `hand_card1_rank` , and `pot_size_ratio` with F scores of 1234, 1199, and 1116 respectively. Logically, this makes sense, as the strength of a person's hand is primarily dependent on what cards they were dealt. Additionally, the importance of the pot size ratio cannot be understated in this context as it determines what stakes are on a specific hand or action and thus exerts major influence on how people play. Features like `prev_preflop_amount` , `suited` , and `positions` are less important but also carry sizeable influence. With our current model, as our focus is more on preflop decision, it also makes sense that `flop_present` , `turn_present` , and `river_present` are of lower importance. This feature importance chart aligns with our expectations of what is important, and is thus an indicator that our model is performing well and properly analyzing relationships in the data to understand what influences hand strength and player decision making.

From the confusion matrix, we can see that our model performed particularly well when predicting when to `Check/Call` and `Bet/Raise` (heavy prominence on the diagonal). It predicted folding perfectly as well, with all **214** samples identified correctly. There were some misclassifications however, as **238** samples of `Check/Call` were misclassified as `Bet/Raise` and, on the other hand, **510** cases of `Bet/Raise` were misclassified as `Check/Call`. As such, roughly **6.23%** of the dataset was misclassified, which is a sizable amount. This error likely is a result of the complicated nature behind poker decisions and the number of features that are involved. Additionally, the feature distributions for certain actions likely overlap, making it difficult for the model to distinguish between the two actions. However, the model does still correctly identify a large majority of the actions and the use of XGBoost in this context is crucial for this as it introduces a new element of non-linearity to capture more complex relationships in the data. Specifically, it's important as poker is a game of extreme strategy and many features that all contribute to actions.

In terms of weaknesses, we might have to reconsider how we weight the categories in the XGBoost model as the `Fold` class is severely underrepresented in terms of samples. We could oversample from this class. Additionally, we also could add more features that are unique to each action or better tune our hyperparameters to prevent the misclassification that is currently occuring. These would help with the recall and precision metrics that we discussed above as well as these misclassifications were the reason for slightly lower scores.

Comparison of the Models

Both of our supervised learning models performed well in predicting poker actions as their misclassification rate was relatively low and had recall and precision values in the low to mid 0.90s. However, the Random Forest Classifier performed slightly better in precision and recall in the `Bet/Raise` class compared to the XGBoost Classifier, showing 0.95 and 0.93 for precision and recall respectively, compared to XGBoost's 0.95 and 0.91. And while it's a slight margin, it shows that the Random Forest Classifier did better in recognizing these more aggressive actions, likely due to it better understanding how different features interacted with each other with its ensemble approach. It also focused on key hand features and we set it up using 100 estimators to increase performance. The XGBoost Classifier used the same feature engineering setup as Random Forest, including pot size ratios, hand card ranks, suited indicators, etc., but might have overfitted due to its emphasis on certain instances that are more aggressive. Additionally, both models had a perfect precision and recall for the `Fold` class, indicating that both were able to distinguish when it is optimal to fold and play less aggressive. While both models did do really well, they both misclassified certain samples between the `Check/Call` and `Bet/Raise` classes (5.65% of the dataset for Random Forest vs. 6.23% for XGBoost), indicating that both models struggled with some feature overlap. A possible method of mitigating this would be to introduce more features that uniquely correspond to certain actions.

Moving aside from the supervised learning models, we also implemented a K-Means clustering algorithm for the purpose of understanding how the data was structured. While we can't directly compare the results of this model to the supervised learning models as we didn't necessarily use it to predict anything, it showed that we could group similar poker hands together based on similarity in features such as bet amounts, pot sizes, and player positions. This approach was rendered valid through the high silhouette score that we computed. This finding was incredibly important in informing our strategy for our supervising model as it helped with the feature engineering process. Additionally, it showed us that most data points fell into clusters representing common moves with less risk, which is in line with poker strategy. Ultimately, we will use a combination of these models to develop a poker bot, drawing from their strengths and what they each highlight.

Next Steps:

Moving forward, we need to incorporate as many features as we can into the models, especially things like pot odds, bet sizes, and opponent player behavioral analysis. All of these are essential to both making the optimal decision with each action throughout a real-world context. Adding these features is also crucial to combatting the misclassification that we faced in our supervised learning models as by adding more features that are unique to certain outcomes, the model can be more sure whether to group something as `Check/Call` or `Bet/Raise`.

While our three models both perform relatively accurately, there are steps that we can take in order to improve the performance of our models. In terms of unsupervised learning, we can apply other classification techniques like hierarchical clustering or DBSCAN (to combat outliers) and examine if that is a better fit for our data than K-Means. We could also improve our Silhouette score for the K-Means algorithm by employing dimensionality reduction techniques to identify non-linear relationships,

In terms of supervised learning, we can improve Random Forest and XGBoost performance by additional feature engineering (adding pot odds, bet sizes, etc.) and using the UCI dataset [7] to calculate hand valuations. We can also tune our hyperparameters, and, more specifically, increase the number of trees, the depth of each tree, and the number of samples we have to see if this improves model performance. Additionally, a major issue that we are currently facing is an underrepresentation of the `Fold` class (only 214 samples), and this data imbalance highlights the need to incorporate class weighting or oversampling.

We will also experiment with using a neural network to improve prediction accuracy and use the NeuronPoker environment [4] to train our bot with reinforcement learning. Ultimately, we want for this neural network to train itself as it goes through and plays games of poker, and we can do this by creating some form of a reward structure where we assign values to certain poker actions. We will then incorporate real-time integration with PyPokerEngine [6] to attain our goal of creating a bot that can replace a human. This would involve creating a UI interface and would necessitate real time decision making. This is the end goal of the project.

Ultimately, we still need to focus on feature engineering by incorporating more statistics (some of which might be real time), model tuning by adjusting hyperparameters, data balancing either through oversampling, class weightage, or SMOTE, and reinforcement learning to have the bot learn by itself through an environment.

References:

[1] V. Lisy and M. Bowling, “Equilibrium Approximation Quality of Current No-Limit Poker Bots”, arXiv, January, 2017. [Online]. Available: https://arxiv.org/pdf/1612.07547

[2] M. Moravčík et al., “DeepStack: Expert-Level Artificial Intelligence in Heads-Up No-Limit Poker”, arXiv, March, 2017. [Online]. Available: https://arxiv.org/pdf/1701.01724

[3] N. Brown and T. Sandholm, “Libratus: The Superhuman AI for No-Limit Poker”, IJCAI-17, August, 2017. [Online]. Available: https://www.ijcai.org/proceedings/2017/0772.pdf

[4] https://github.com/dickreuter/neuron_poker

[5] https://www.science.org/doi/10.1126/science.aay2400

[6] https://github.com/ishikota/PyPokerEngine

[7] https://archive.ics.uci.edu/dataset/158/poker+hand

GANTT Chart

Click here to view our GANTT Chart

Project Contribution Chart

Name	Final Report Contribution
Siddharth	Introduction/Background, Problem Definition, Video Presentation
Aneesh	Introduction/Background, Problem Definition, Model Analysis, Model Comparison, Next Steps, GANTT Chart
Vishal	Video Presentation
Aaryam	Supervised Learning Method Implementation - XGBoost
Jay	Supervised Learning Method Implementation - XGBoost

Video Presentation