STAT 441 Final Project Appendix

Mark Chen, Reven Liu, Arvind Nagabhirava, Rain Zhao

2023-04-24

Contents

1 Exploratory Data Analysis

```
import pandas as pd
import numpy as np

# Plotting
import matplotlib.pyplot as plt

from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

import random
import tensorflow as tf

# Calibration time
import time
```

```
nba = pd.read_csv(
  '/kaggle/input/nba-time-fixed/NBA Dataset Time Fixed - NBA_Aggregated_Dataset (1).csv')

nba['POS'].fillna('C', inplace=True) # Make Eddy Curry a Center
nba = nba.drop(nba[nba.POS == 'GF'].index) # Remove Jiri Welsch

positions = {'SG' : 0, 'PG': 0, 'G' : 0,
             'C' : 1, 'SF' : 1, 'F': 1, 'PF': 1}
```

```
# Assign every player into one of two classes
nba_binary = nba.replace({"POS": positions})
```

```
def years_since(season):
    season = int(season.split("-")[0])
    return season-2001

nba_binary["years_since"] = nba_binary["SEASON"].apply(lambda x: years_since(x))
# Convert Factor: "2001-2002" to Int: 2001
nba_binary = nba_binary[nba_binary['SEASON'] != '2011-2012'] # Remove the lockout season

nba_binary["2PA"] = nba_binary["FGA"] - nba_binary["3PA"]
nba_binary["2PM"] = nba_binary["FGM"] - nba_binary["3PM"]
nba_binary["2P%"] = nba_binary["2PM"]/nba_binary["2PA"]
nba_binary['2P%'] = nba_binary['2P%'].fillna(0)
nba_binary['2P%'] = nba_binary['2P%'].clip(upper=1)
```

```
nba_forward = nba_binary[nba_binary['POS'] == 1]
nba_guard = nba_binary[nba_binary['POS'] == 0]
```

```
import seaborn as sns

numeric_features = nba_binary.columns[3:]
nfeatures = len(numeric_features)
ncol = np.sqrt(nfeatures).astype(int)+1
nrow = ncol
fig, axs = plt.subplots(nrow, ncol, figsize=(5*ncol, 5*nrow))

for r in range(nrow):
    for c in range(ncol):
        if ncol*r+c < nfeatures:
            feature = numeric_features[ncol*r+c]
            f_data = nba_forward[feature]
            g_data = nba_guard[feature]

            f_mean = f_data.mean() # Compute the mean for forward players
            g_mean = g_data.mean() # Compute the mean for guard players

            sns.kdeplot(f_data, color='r', label='Forward', ax=axs[r,c])
            sns.kdeplot(g_data, color='g', label='Guard', ax=axs[r,c])

            axs[r, c].set_title(feature)
            axs[r, c].set_xlabel('Value')
            axs[r, c].set_ylabel('Density')
            axs[r, c].legend()

plt.tight_layout()
plt.show()
```

```
import seaborn as sns

numeric_features = ["FGA", "FG%", "3PA", "3P%", "FT%", "REB", "AST", "BLK"]
nfeatures = len(numeric_features)
ncol = 4
```

```
nrow = 2
fig, axs = plt.subplots(nrow, ncol, figsize=(5*ncol, 5*nrow))

for r in range(nrow):
    for c in range(ncol):
        if ncol*r+c < nfeatures:
            feature = numeric_features[ncol*r+c]
            f_data = nba_forward[feature]
            g_data = nba_guard[feature]

            f_mean = f_data.mean() # Compute the mean for forward players
            g_mean = g_data.mean() # Compute the mean for guard players

            sns.kdeplot(f_data, color='r', label='Forward', ax=axs[r,c])
            sns.kdeplot(g_data, color='g', label='Guard', ax=axs[r,c])

            axs[r, c].set_title(feature)
            axs[r, c].set_xlabel('Value')
            axs[r, c].set_ylabel('Density')
            axs[r, c].legend()

plt.tight_layout()
plt.show()
```

2 Support Vector Machines and Bagging

```
import pandas as pd
import numpy as np
# Plotting
import matplotlib.pyplot as plt
import seaborn as sns
# SVM
from sklearn import svm
from sklearn.svm import LinearSVC
# Tree
from sklearn import tree
# Bagging
from sklearn.ensemble import BaggingClassifier
# Splitting, cross validation, pipelines
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, KFold
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.inspection import permutation_importance
from sklearn.compose import ColumnTransformer, make_column_selector
# Metrics
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Calibration time
import time
```

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

```
# Read in the data
nba = pd.read_csv('/content/drive/My Drive/NBA_Aggregated_Dataset.csv')
```

The NBA dataset contains statistics on NBA players from 2001 to 2022. Features include games played, points per game, steals per game, etc. The problem of interest is to classify players' positions based on their stats.

```
nba.head()
```

```
nba[nba['POS'].isnull()]
```

The target variable is POS (position). The only missing value for POS is for Eddy Curry, who, with a bit of research, can confirm plays as C (center).

```
nba['POS'].fillna('C', inplace=True)
```

The different positions are as follows:
- SG: shooting guard - C: center - SF: small forward - G: guard ? - F: forward ? - PF: power forward - PG: point guard - GF: guard/forward. The classification problem of interest is to predict whether players are a forward (SF, PF, or C) or a guard (PG, SG).
Let 1 represent forwards and 0 represent guards. To avoid ambiguity about guard forwards, who can play both positions, we will drop rows with 'GF' values. This only removes 4 rows corresponding to seasons for the single GF player Jiri Welsch.

```
nba = nba.drop(nba[nba.POS == 'GF'].index)
```

```
positions = {'SG' : 0, 'C' : 1, 'SF' : 1, 'G' : 0,
             'F': 1, 'PF': 1, 'PG': 0}
```

```
# Assign every player into one of two classes
nba_binary = nba.replace({"POS": positions})
nba_binary['POS'].unique()
```

```
nba_binary.head()
```

```
# Drop the excess columns
list(nba_binary)[22:43]
nba_binary = nba_binary.drop(list(nba_binary)[22:43], axis=1)
```

```
nba_binary = nba_binary[nba_binary['SEASON'] != '2011-2012'] # Remove the lockout season
nba_binary['POS'].value_counts() # 5666 forwards, 3923 guards (2/3 imbalance)
```

```
# Feature engineering
nba_binary["2PA"] = nba_binary["FGA"] - nba_binary["3PA"]
nba_binary["2PM"] = nba_binary["FGM"] - nba_binary["3PM"]
nba_binary["2P%"] = nba_binary["2PM"]/nba_binary["2PA"]
nba_binary['2P%'] = nba_binary['2P%'].fillna(0)
nba_binary['2P%'] = nba_binary['2P%'].clip(upper=1)
```

```
nba_forward = nba_binary[nba_binary['POS'] == 1]
nba_guard = nba_binary[nba_binary['POS'] == 0]
nba_forward.shape, nba_guard.shape
```

Below we have the test data set for 2022-2023:

```
nba_recent = pd.read_csv('/content/drive/My Drive/2022-2023_NBA_Data.csv')
```

```
# Assign every player into one of two classes
nba_recent = nba_recent.replace({"POS": positions})
nba_recent['POS'].unique()
```

```
nba_recent["2PA"] = nba_recent["FGA"] - nba_recent["3PA"]
nba_recent["2PM"] = nba_recent["FGM"] - nba_recent["3PM"]
nba_recent["2P%"] = nba_recent["2PM"]/nba_recent["2PA"]
nba_recent['2P%'] = nba_recent['2P%'].fillna(0)
nba_recent['2P%'] = nba_recent['2P%'].clip(upper=1)
```

The below code performs transformations necessary to generate the radial plots in plotly:

```
import plotly.express as px
df = nba_binary.copy()
df['GP_PCT']  = df['GP']/max(df['GP'])
df['MIN_PCT'] = df['MIN']/max(df['MIN'])
df['PTS_PCT'] = df['PTS']/max(df['PTS'])
df['FGM_PCT'] = df['FGM']/max(df['FGM'])
df['FGA_PCT'] = df['FGA']/max(df['FGA'])
df['FG%_PCT'] = df['FG%']/max(df['FG%'])
df['3PM_PCT'] = df['3PM']/max(df['3PM'])
df['3PA_PCT'] = df['3PA']/max(df['3PA'])
df['3P%_PCT'] = df['3P%']/max(df['3P%'])
df['FTM_PCT'] = df['FTM']/max(df['FTM'])
df['FTA_PCT'] = df['FTA']/max(df['FTA'])
df['FT%_PCT'] = df['FT%']/max(df['FT%'])
df['REB_PCT'] = df['REB']/max(df['REB'])
df['AST_PCT'] = df['AST']/max(df['AST'])
```

```
df['STL_PCT'] = df['STL']/max(df['STL'])
df['BLK_PCT'] = df['BLK']/max(df['BLK'])
df['TO_PCT']  = df['TO']/max(df['TO'])
df['DD2_PCT'] = df['DD2']/max(df['DD2'])
df['TD3_PCT'] = df['TD3']/max(df['TD3'])
column_names = list(df.columns.values)
#column_names
agg_radar_df = df[['POS', 'REB_PCT', 'AST_PCT', 'PTS_PCT', 'BLK_PCT', '3PM_PCT']]
agg_radar_df
```

```
full_grouped_df = df[['POS', 'GP_PCT', 'MIN_PCT', 'PTS_PCT', 'FGM_PCT', 'FGA_PCT',
                      'FG%_PCT', '3PM_PCT', '3PA_PCT', '3P%_PCT', 'FTM_PCT',
                      'FTA_PCT','FT%_PCT', 'REB_PCT', 'AST_PCT', 'STL_PCT',
                      'BLK_PCT', 'TO_PCT', 'DD2_PCT', 'TD3_PCT']]
full_grouped_df = full_grouped_df.groupby(['POS']).mean()
full_grouped_df
```

```
plotly_df = agg_radar_df.groupby(['POS']).mean()
plotly_df
```

```
guard_player = pd.DataFrame(dict(
    stats=plotly_df.iloc[0].to_numpy(),
    features=['REB_PCT', 'AST_PCT', 'PTS_PCT', 'BLK_PCT', '3PM_PCT']))
fig = px.line_polar(guard_player, r='stats', theta='features', line_close=True)
fig.update_layout(title='Guard Statistics')
fig.show()
```

```
forward_player = pd.DataFrame(dict(
    stats=plotly_df.iloc[1].to_numpy(),
    features=['REB_PCT', 'AST_PCT', 'PTS_PCT', 'BLK_PCT', '3PM_PCT']))
fig2 = px.line_polar(forward_player, r='stats', theta='features', line_close=True)
fig.update_layout(title='Forward Statistics')
fig2.show()
```

2.1 SVM

```
# Exclude POS, NAME, SEASON, and year from the training set
nba_X = nba_binary.drop(columns = ["POS", "NAME", "SEASON"]).to_numpy()
nba_y = nba_binary["POS"].to_numpy()
```

```
# Training data
train_X = nba_X
train_y = nba_y
```

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif

# Mutual information
selector = SelectKBest(mutual_info_classif, k=10)
X_new = selector.fit_transform(train_X, train_y)

selected_features = selector.get_support(indices=True)

print("Selected feature indices:", selected_features)
```

```
names = ["GP", "FGM", "FG%", "3PM", "3PA", "3P%", "FT%", "REB", "AST",
"STL", "BLK", "TO", "DD2", "TD3", "2PM", "2P%"]
mutual_X = nba_binary.drop(columns = ["POS", "NAME", "SEASON", "MIN",
"PTS", "FTM", "FTA", "2PA"]).to_numpy()
mutual_y = nba_binary["POS"].to_numpy()
mutual_info = mutual_info_classif(mutual_X, mutual_y, random_state=42)

# Plot of mutual information
fig, ax = plt.subplots()
ax.bar(range(mutual_X.shape[1]), mutual_info)
ax.set_xticks(range(mutual_X.shape[1]))
ax.set_xticklabels(nba_binary.drop(columns = ["POS", "NAME",
"SEASON", "MIN", "PTS", "FTM", "FTA", "2PA"]), rotation=90)
ax.set_xlabel("Feature")
ax.set_ylabel("Mutual Information")
ax.set_title("Mutual Information Scores for NBA players")
plt.show()
```

```
column_names = nba_binary.columns.values[3:]
svm_f = column_names[selected_features]
svm_f
exclude = list(set(column_names) - set(svm_f)) #excluded columns
```

```
exclude = exclude + ["POS", "NAME", "SEASON"]
nba_X = nba_binary.drop(columns = exclude).to_numpy()
nba_y = nba_binary["POS"].to_numpy()
train_X = nba_X
train_y = nba_y
```

Grid search cross validation for the linear kernel svm:

```
start_time = time.perf_counter()

model_linear = Pipeline(
    steps=[("scaler", StandardScaler()),
           ("model", LinearSVC(class_weight = 'balanced', dual = False,
                               random_state = 441))]
)

cv = KFold(
    n_splits=5,
    shuffle=False
)

svm_grid = {'model__C': [.01, 0.1, 1, 10]}
            #"model__class_weight": [None, "balanced"]}

cv_svm = GridSearchCV(
    estimator = model_linear,
    param_grid = svm_grid,
    n_jobs = -1,
    cv = cv,
    verbose = 4
)
```

```
cv_svm.fit(X = train_X, y = train_y)

end_time = time.perf_counter()

runtime = end_time - start_time

print(f"Runtime: {runtime:.6f} seconds")
```

```
cv_svm.best_params_
```

```
test_X = nba_recent.drop(columns = exclude).to_numpy()
test_y = nba_recent["POS"].to_numpy()
```

```
model_linear.fit(X = train_X, y = train_y)
print("Linear-CV model train error: {}".format(
    1.0 - model_linear.score(X=train_X, y=train_y)
))
print("Linear-CV model test error: {}".format(
    1.0 - model_linear.score(X=test_X, y=test_y)
))
```

Grid search cross validation for the rbf kernel svm:

```
start_time = time.perf_counter()

model_rbf = Pipeline(
    steps=[("scaler", StandardScaler()),
           ("model", svm.SVC(kernel="rbf", class_weight = 'balanced',
                             random_state = 441))]
)

cv = KFold(
    n_splits=5,
    shuffle=False
)

rbf_svm_grid = {'model__C': [.01, 0.1, 1, 10],
            'model__gamma': ['scale', 'auto']}

cv_svm_rbf = GridSearchCV(
    estimator = model_rbf,
    param_grid = rbf_svm_grid,
    n_jobs = -1,
    cv = cv,
    verbose = 4
)

cv_svm_rbf.fit(X = train_X, y = train_y)

end_time = time.perf_counter()

runtime = end_time - start_time

print(f"Runtime: {runtime:.6f} seconds")
```

```
cv_svm_rbf.best_params_
```

```
model_rbf.fit(train_X, train_y)
print("RBF-CV model train error: {}".format(
    1.0 - model_rbf.score(X=train_X, y=train_y)
))
print("RBF-CV model test error: {}".format(
    1.0 - model_rbf.score(X=test_X, y=test_y)
))
```

Grid search cross validation for the polynomial kernel svm:

```
start_time = time.perf_counter()

model_poly = Pipeline(
    steps=[("scaler", StandardScaler()),
           ("model", svm.SVC(kernel="poly", class_weight = 'balanced',
                             random_state = 41))]
)

poly_cv = KFold(
    n_splits=5,
    shuffle=False
)

poly_svm_grid = {'model__C': [.01, 0.1, 1, 10],
            'model__gamma': ['scale', 'auto']}

cv_svm_poly = GridSearchCV(
    estimator = model_poly,
    param_grid = poly_svm_grid,
    n_jobs = -1,
    cv = poly_cv,
    verbose = 4
)

cv_svm_poly.fit(X = train_X, y = train_y)

end_time = time.perf_counter()

runtime = end_time - start_time

print(f"Runtime: {runtime:.6f} seconds")
```

```
cv_svm_poly.best_params_
```

```
model_poly.fit(train_X, train_y)
print("RBF-CV model train error: {}".format(
    1.0 - model_poly.score(X=train_X, y=train_y)
))
print("RBF-CV model test error: {}".format(
    1.0 - model_poly.score(X=test_X, y=test_y)
))
```

Confusion Matrices:

```
# linear
y_pred_linear_svm = model_linear.predict(test_X)
cm_linear_svm = confusion_matrix(test_y, y_pred_linear_svm)
positions = ["Guard", "Forward"]
svm_linear_matrix = ConfusionMatrixDisplay(cm_linear_svm, display_labels = positions).plot()
```

```
# RBF
y_pred_svm = model_rbf.predict(test_X)
cm_svm = confusion_matrix(test_y, y_pred_svm)
positions = ["Guard", "Forward"]
svm_c_matrix = ConfusionMatrixDisplay(cm_svm, display_labels = positions).plot()
```

```
# Poly
y_pred_poly_svm = model_poly.predict(test_X)
cm_poly_svm = confusion_matrix(test_y, y_pred_poly_svm)
positions = ["Guard", "Forward"]
svm_poly_matrix = ConfusionMatrixDisplay(cm_poly_svm, display_labels = positions).plot()
```

2.2 Bagging

Bagging uses the DecisionTreeClassifier() as the base estimator, which we can perform parameter calibration on using cross validation.

```
start_time = time.perf_counter()

tree_clf = tree.DecisionTreeClassifier(random_state = 41)
tree_grid = {'criterion': ['gini', 'entropy', 'log_loss'],
               'max_depth': [10, 20, 30, 40, 50],
               'min_samples_split': [2, 4,6,8],
               'min_samples_leaf': [1,2,3],
               'max_features': ['sqrt', 'log2']}

random_search = RandomizedSearchCV(estimator = tree_clf, param_distributions = tree_grid,
                                   n_iter = 100, cv = 5, verbose=4, n_jobs = -1,
                                   random_state = 41)
random_search.fit(train_X, train_y)
random_search.best_params_
end_time = time.perf_counter()

runtime = end_time - start_time

print(f"Runtime: {runtime:.6f} seconds")
```

```
random_search.best_params_
```

```
tuned_tree = tree.DecisionTreeClassifier(
    criterion = 'log_loss',
    min_samples_split = 8,
    min_samples_leaf = 1,
    max_features = 'log2',
    max_depth = 10,
    random_state = 41
)

tuned_bagging = BaggingClassifier(estimator = tuned_tree, random_state = 41)
```

```
tuned_bagging.fit(train_X, train_y)
```

Confusion Matrix:

```
y_pred = tuned_bagging.predict(test_X)
```

```
cm = confusion_matrix(test_y, y_pred)
```

```
positions = ["Guard", "Forward"]
c_matrix = ConfusionMatrixDisplay(cm, display_labels = positions).plot()
```

```
# Bagging results
print("Bagging-CV model train error: {}".format(
    1.0 - tuned_bagging.score(X=train_X, y=train_y)
))
print("Bagging-CV model test error: {}".format(
    1.0 - tuned_bagging.score(X=test_X, y=test_y)
))
```

3 Neural Networks

```
import pandas as pd
import numpy as np

# Plotting
import matplotlib.pyplot as plt

from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

import random
import tensorflow as tf

# Calibration time
import time
```

```
nba = pd.read_csv("/kaggle/input/all-nba/All NBA Data.csv")
nba['POS'].fillna('C', inplace=True) # Make Eddy Curry a Center
nba = nba.drop(nba[nba.POS == 'GF'].index) # Remove Jiri Welsch

positions = {'SG' : 0, 'PG': 0, 'G' : 0,
             'C' : 1, 'SF' : 1, 'F': 1, 'PF': 1}

# Assign every player into one of two classes
nba_binary = nba.replace({"POS": positions})

def years_since(season):
    season = int(season.split("-")[0])
    return season-2001

nba_binary["years_since"] = nba_binary["SEASON"].apply(lambda x: years_since(x))
# Convert Factor: "2001-2002" to Int: 2001
nba_binary = nba_binary[nba_binary['SEASON'] != '2011-2012'] # Remove the lockout season

nba_binary["2PA"] = nba_binary["FGA"] - nba_binary["3PA"]
nba_binary["2PM"] = nba_binary["FGM"] - nba_binary["3PM"]
nba_binary["2P%"] = nba_binary["2PM"]/nba_binary["2PA"]
nba_binary['2P%'] = nba_binary['2P%'].fillna(0)
nba_binary['2P%'] = nba_binary['2P%'].clip(upper=1)
```

```
# Create NBA_Train and NBA_Test dataframes
NBA_Train = nba_binary[nba_binary['SEASON'] != "2022-2023"]
NBA_Test = nba_binary[nba_binary['SEASON'] == "2022-2023"]

NBA_Train_X = NBA_Train.drop(columns = ["POS", "NAME", "SEASON"]).to_numpy()
NBA_Train_y = NBA_Train["POS"].to_numpy()

NBA_Test_X = NBA_Test.drop(columns = ["POS", "NAME", "SEASON"]).to_numpy()
NBA_Test_y = NBA_Test["POS"].to_numpy()
```

```
nba_forward = NBA_Train[NBA_Train['POS'] == 1]
nba_guard = NBA_Train[NBA_Train['POS'] == 0]
```

Cross Validation for Number of Hidden Layers

```
start_time = time.perf_counter()

X = NBA_Train_X
y = NBA_Train_y

k = 5
kf = KFold(n_splits=k, shuffle=True)

one_hidden_acc = []
two_hidden_acc = []
three_hidden_acc = []

for train_index, val_index in kf.split(X):
    # split the data into training and validation sets
    X_train, y_train = X[train_index], y[train_index]
    X_val, y_val = X[val_index], y[val_index]

    # One Hidden Layer
    model = Sequential()
    model.add(Dense(16, activation='relu', input_shape=(23,)))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    model.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)

    loss, acc = model.evaluate(X_val, y_val, verbose=0)
    one_hidden_acc.append(acc)

    # Two Hidden Layers
    model2 = Sequential()
    model2.add(Dense(16, activation='relu', input_shape=(23,)))
    model2.add(Dense(32, activation='relu'))
    model2.add(Dense(1, activation='sigmoid'))
    model2.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    model2.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)

    loss, acc = model2.evaluate(X_val, y_val, verbose=0)
    two_hidden_acc.append(acc)

    # Three Hidden Layer
    model3 = Sequential()
    model3.add(Dense(16, activation='relu', input_shape=(23,)))
    model3.add(Dense(32, activation='relu'))
    model3.add(Dense(32, activation='relu'))
    model3.add(Dense(1, activation='sigmoid'))
    model3.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    model3.fit(X_train, y_train, epochs=20, batch_size=23, verbose=0)

    loss, acc = model3.evaluate(X_val, y_val, verbose=0)
    three_hidden_acc.append(acc)
```

```
# Performance is nearly identical, use one hidden layer by occam's razor
print(f"Average accuracy with one hidden layer {np.mean(one_hidden_acc)}")
print(f"Average accuracy with two hidden layers {np.mean(two_hidden_acc)}")
print(f"Average accuracy with three hidden layers {np.mean(three_hidden_acc)}")

end_time = time.perf_counter()
runtime = end_time - start_time
print(f"Runtime: {runtime:.6f} seconds")
```

Cross Validation for Number of Neurons in Hidden Layer

```
start_time = time.perf_counter()

neurons_24_acc = []
neurons_48_acc = []
neurons_96_acc = []

# loop over the folds
for train_index, val_index in kf.split(X):
    # split the data into training and validation sets
    X_train, y_train = X[train_index], y[train_index]
    X_val, y_val = X[val_index], y[val_index]

    # define and compile the Keras model
    model = Sequential()
    model.add(Dense(24, activation='relu', input_shape=(23,)))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    # train the model on the training set
    model.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)

    # evaluate the model on the validation set
    loss, acc = model.evaluate(X_val, y_val, verbose=0)
    neurons_24_acc.append(acc)

    # define and compile the Keras model
    model2 = Sequential()
    model2.add(Dense(48, activation='relu', input_shape=(23,)))
    model2.add(Dense(1, activation='sigmoid'))
    model2.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    # train the model on the training set
    model2.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)

    # evaluate the model on the validation set
    loss, acc = model2.evaluate(X_val, y_val, verbose=0)
    neurons_48_acc.append(acc)

    # define and compile the Keras model
    model3 = Sequential()
    model3.add(Dense(96, activation='relu', input_shape=(23,)))
    model3.add(Dense(1, activation='sigmoid'))
    model3.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
    # train the model on the training set
    model3.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)

    # evaluate the model on the validation set
    loss, acc = model3.evaluate(X_val, y_val, verbose=0)
    neurons_96_acc.append(acc)

# Again, performance is nearly identical so we use 24 neurons
print(f"Average accuracy with 24 neurons {np.mean(neurons_24_acc)}")
print(f"Average accuracy with 48 neurons {np.mean(neurons_48_acc)}")
print(f"Average accuracy with 96 neurons {np.mean(neurons_96_acc)}")

end_time = time.perf_counter()
runtime = end_time - start_time
print(f"Runtime: {runtime:.6f} seconds")
```

Model Fitting

```
# Actual Model
start_time = time.perf_counter()

seed = 25
np.random.seed(seed)
random.seed(seed)
tf.random.set_seed(seed)

nba_model = Sequential()
nba_model.add(Dense(24, activation='relu', input_shape=(23,)))
nba_model.add(Dense(1, activation='sigmoid'))
nba_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# train the model on the training set
nba_model.fit(NBA_Train_X, NBA_Train_y, epochs=100, batch_size=32, verbose=0)

# evaluate the model on the validation set
loss, nba_acc = nba_model.evaluate(NBA_Test_X, NBA_Test_y, verbose=1)

print(f"{seed}: {1-nba_acc}")

end_time = time.perf_counter()
runtime = end_time - start_time
print(f"Runtime: {runtime:.6f} seconds")
```

4 Hybrid Model

```
import pandas as pd
import numpy as np
# Plotting
import matplotlib.pyplot as plt
import seaborn as sns
# SVM
from sklearn import svm
from sklearn.svm import LinearSVC
# Tree
from sklearn import tree
# Logistic Regression
from sklearn.linear_model import LogisticRegression
# Bagging
from sklearn.ensemble import BaggingClassifier
# Splitting, cross validation, pipelines
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, KFold
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.inspection import permutation_importance
from sklearn.compose import ColumnTransformer, make_column_selector
from sklearn.inspection import PartialDependenceDisplay
# Metrics
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
# Calibration time
import time
```

```
nba = pd.read_csv('NBA_Aggregated_Dataset.csv')
```

The NBA dataset contains statistics on NBA players from 2001 to 2022. Features include games played, points per game, steals per game, etc. The problem of interest is to classify players' positions based on their stats.

```
nba.head()
```

```
nba.info()
```

It looks like reading in the csv includes some empty columns from Excel, which we will want to drop later.

```
nba['POS'].unique()
```

```
nba[nba['POS'].isnull()]
```

The target variable is POS (position). The only missing value for POS is for Eddy Curry, who, with a bit of research, can confirm plays as C (center).

```
nba['POS'].fillna('C', inplace=True)
```

```
nba['POS'].unique()
```

```
nba = nba.drop(nba[nba.POS == 'GF'].index)
```

```
positions = {'SG' : 0, 'C' : 1, 'SF' : 1, 'G' : 0,
             'F': 1, 'PF': 1, 'PG': 0}

# Assign every player into one of two classes
```

```
nba_binary = nba.replace({"POS": positions})
nba_binary['POS'].unique()
```

```
nba_binary.head()
```

```
# Drop the excess columns
list(nba_binary)[22:43]
nba_binary = nba_binary.drop(list(nba_binary)[22:43], axis=1)
```

```
nba_binary.describe()
```

```
nba_binary = nba_binary[nba_binary['SEASON'] != '2011-2012'] # Remove the lockout season
nba_binary['POS'].value_counts() # 5666 forwards, 3923 guards (2/3 imbalance)
```

```
# Feature engineering
nba_binary["2PA"] = nba_binary["FGA"] - nba_binary["3PA"]
nba_binary["2PM"] = nba_binary["FGM"] - nba_binary["3PM"]
nba_binary["2P%"] = nba_binary["2PM"]/nba_binary["2PA"]
nba_binary['2P%'] = nba_binary['2P%'].fillna(0)
nba_binary['2P%'] = nba_binary['2P%'].clip(upper=1)
```

```
nba_binary.head
```

We propose the following classifiers: - Support Vector Machines - Random Forests - Combined SVM with tree bagging

```
nba_binary.shape
```

```
nba_forward = nba_binary[nba_binary['POS'] == 1]
nba_guard = nba_binary[nba_binary['POS'] == 0]
nba_forward.shape, nba_guard.shape
```

```
nba_recent = pd.read_csv('2022-2023_NBA_Data.csv')
```

```
nba_recent['POS'].unique()
```

```
nba_recent[nba_recent['POS'].isnull()]
```

```
nba_recent.info
```

```
nba_recent.head()
```

```
# Assign every player into one of two classes
nba_recent = nba_recent.replace({"POS": positions})
nba_recent['POS'].unique()
```

```
nba_recent['POS'].value_counts() # 295 forwards, 239 guards
```

```
nba_recent["2PA"] = nba_recent["FGA"] - nba_recent["3PA"]
nba_recent["2PM"] = nba_recent["FGM"] - nba_recent["3PM"]
nba_recent["2P%"] = nba_recent["2PM"]/nba_recent["2PA"]
nba_recent['2P%'] = nba_recent['2P%'].fillna(0)
nba_recent['2P%'] = nba_recent['2P%'].clip(upper=1)
```

Feature Selection:

```
nba_X = nba_binary.drop(columns = ["POS", "NAME", "SEASON"])
nba_y = nba_binary["POS"]
```

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif

# Compute mutual information between each feature and target variable
mutual_info = mutual_info_classif(nba_X, nba_y, random_state=42)

# Plot mutual information scores for each feature
fig, ax = plt.subplots()
ax.bar(range(nba_X.shape[1]), mutual_info)
ax.set_xticks(range(nba_X.shape[1]))
ax.set_xticklabels(nba_X.columns, rotation=90)
ax.set_xlabel("Feature")
ax.set_ylabel("Mutual Information")
ax.set_title("Mutual Information Scores for NBA players")
plt.show()
```

```
# Perform feature selection with mutual information
selector = SelectKBest(mutual_info_classif, k=10)
X_new = selector.fit_transform(nba_X, nba_y)

# Get the indices of the selected features
selected_features_indices = selector.get_support(indices=True)

# Print the indices of the selected features
print("Indices of selected features:", selected_features_indices)
selected_features = nba_X.columns[selected_features_indices]
selected_features
```

```
train_X = nba_binary[selected_features].to_numpy()
train_y = nba_binary["POS"].to_numpy()
test_X = nba_recent[selected_features].to_numpy()
test_y = nba_recent["POS"].to_numpy()
```

A single Decision Tree model

Let's visualize the splits of a decision tree trained on the 2 most important features found by bagging: REB and AST.

```
tuned_tree = tree.DecisionTreeClassifier(
    criterion = 'log_loss',
    min_samples_split = 8,
    min_samples_leaf = 1,
    max_features = 'log2',
    max_depth = 10,
    random_state = 41
)

tuned_bagging = BaggingClassifier(estimator = tuned_tree, random_state = 41)
tuned_bagging.fit(train_X, train_y)
```

```
from sklearn.inspection import DecisionBoundaryDisplay
import matplotlib.patches as mpatches
```

```
reb_ast_axes = np.where(np.logical_or(selected_features == 'REB', selected_features == 'AST'))[0]
tuned_tree.fit(train_X[:,reb_ast_axes], train_y)
plt.figure(figsize=(20,10))
```

```
tree.plot_tree(tuned_tree, max_depth=2)
```

```
y_pred_tree = tuned_tree.predict(train_X[:,reb_ast_axes])
print("Decision Tree train error: {}".format(
    np.mean(y_pred_tree != train_y)
))
```

```
y_pred_tree = tuned_tree.predict(test_X[:,reb_ast_axes])
print("Decision Tree test error: {}".format(
    np.mean(y_pred_tree != test_y)
))
```

```
def plotDecisionBoundary2D(model, X, y, **predict_params):
    feature_1, feature_2 = np.meshgrid(
        np.linspace(X[:, 0].min(), X[:, 0].max(), num=400),
        np.linspace(X[:, 1].min(), X[:, 1].max(), num=400)
    )
    grid = np.vstack([feature_1.ravel(), feature_2.ravel()]).T
    y_pred = np.reshape(model.predict(grid, **predict_params), feature_1.shape)
    display = DecisionBoundaryDisplay(
        xx0=feature_1, xx1=feature_2, response=y_pred
    )
    display.plot()
    yellow_patch = mpatches.Patch(color='yellow', label='forward')
    purple_patch = mpatches.Patch(color='purple', label='guard')
    display.ax_.legend(handles=[yellow_patch, purple_patch])
    display.plot()
    display.ax_.scatter(X[:, 0], X[:, 1], c=y, edgecolor="black", alpha=0.5)
    return display
```

```
display = plotDecisionBoundary2D(tuned_tree, train_X[:,reb_ast_axes], train_y)
display.ax_.set(xlabel='REB', ylabel='AST')
plt.show()
```

Now visualize the first 2 splits of a decision tree trained on the full data:

```
tuned_tree.fit(train_X, train_y)
plt.figure(figsize=(10,5))
tree.plot_tree(tuned_tree, max_depth=1)
```

```
y_pred_tree = tuned_tree.predict(train_X)
print("Decision Tree train error: {}".format(
    np.mean(y_pred_tree != train_y)
))
```

```
y_pred_tree = tuned_tree.predict(test_X)
print("Decision Tree test error: {}".format(
    np.mean(y_pred_tree != test_y)
))
```

```
fig, ax = plt.subplots(1, 3, figsize=(3*5, 1*5))
ax[0].scatter(nba_forward['3PM'], nba_forward['FGA'], alpha=0.1, label='forward')
ax[0].scatter(nba_guard['3PM'], nba_guard['FGA'], alpha=0.1, label='guard')
ax[0].axvline(x = 0.05, color='r', label = '3PM split')
ax[0].legend()
ax[0].set_title('First split')
```

```
ax[0].set(xlabel='3PM', ylabel='FGA')

ax[1].scatter(nba_forward[nba_forward['3PM'] <= 0.05]['FGA'],
nba_forward[nba_forward['3PM'] <= 0.05]['3PM'], alpha=0.1, label='forward')
ax[1].scatter(nba_guard[nba_guard['3PM'] <= 0.05]['FGA'],
nba_guard[nba_guard['3PM'] <= 0.05]['3PM'], alpha=0.1, label='guard')
ax[1].axvline(x = 2.65, color='r', label = 'FGA split')
ax[1].legend()
ax[1].set_title('Second split')
ax[1].set(xlabel='FGA', ylabel='3PM')

ax[2].scatter(nba_forward[nba_forward['3PM'] > 0.05]['3PA'],
nba_forward[nba_forward['3PM'] > 0.05]['3PM'], alpha=0.1, label='forward')
ax[2].scatter(nba_guard[nba_guard['3PM'] > 0.05]['3PA'],
nba_guard[nba_guard['3PM'] > 0.05]['3PM'], alpha=0.1, label='guard')
ax[2].axvline(x = 0.75, color='r', label = '3PA split')
ax[2].legend()
ax[2].set_title('Second split')
ax[2].set(xlabel='3PA', ylabel='3PM')
```

Hybrid SVM and Tree model

Construct a classification tree using SVM hyerplane splits of the data at each node. The stopping criterion is when the misclassification error in a region is at most `max_miss` or the size of a region doesn't change. All SVM splits use the same parameter `C`. Boosting is performed on misclassified points at each node of the tree except the root. When boosting, misclassified points are assigned weight `boost_strength` * (1 + proportion of correctly classified points).

```
def project(v, u):
    return np.dot(v, u) / np.dot(u, u) * u

def rotation_mat2D(v1, v2):
    return np.c_[v1/np.linalg.norm(v1), v2/np.linalg.norm(v2)].T

def standardized_svm(kernel, **kwargs):
    if kernel == 'linear': # LinearSVC faster than SVC
        return Pipeline(
            steps=[("scaler", StandardScaler()),
                   ("model", svm.LinearSVC(class_weight='balanced', dual=False,
                   max_iter=10000, random_state=41, **kwargs))]
        )
    return Pipeline(
        steps=[("scaler", StandardScaler()),
               ("model", svm.SVC(kernel=kernel, class_weight='balanced',
               random_state = 41, **kwargs))]
    )

class SVMTree:
    def __init__(self, kernel, C, max_miss, boost_strength, depth=0):
        self.kernel = kernel
        self.C = C
        self.max_miss = max_miss
        self.boost_strength = boost_strength
        self.depth = depth
        self.height = 0
```

```
        self.model_svm = None
        self.parent = None
        self.pos = None
        self.neg = None
        self.X = None
        self.y = None

    def fit_svm(self, X, y, sample_weight):
#         print("Begin SVM")
        model_svm = standardized_svm(kernel=self.kernel, C=self.C)
#         try:
#             model_svm = standardized_svm(kernel=self.kernel)
#             cv = KFold(
#                 n_splits=5,
#                 shuffle=True
#             )
#             svm_grid = {'model__C': [0.1, 1, 10, 100, 1000]}
#             cv_svm = GridSearchCV(
#                 estimator = model_svm,
#                 param_grid = svm_grid,
#                 n_jobs = -1,
#                 cv = cv,
#                 verbose = 4
#             )
#             cv_svm.fit(X, y, model__sample_weight=sample_weight)
#             print(cv_svm.best_params_)
#             model_svm = standardized_svm(
#                 kernel=self.kernel,
#                 C=cv_svm.best_params_['model__C'])
#         except:
#             print("Not enough samples for CV, falling back to C=1")
#             model_svm = standardized_svm(kernel=self.kernel)
        model_svm.fit(X, y, model__sample_weight=sample_weight)
        self.svm = model_svm
        return model_svm

    def fit(self, X, y, sample_weight=None):
#         print(f"level {self.depth}")
        self.X = X
        self.y = y
        self.fit_svm(X, y, sample_weight)
        pos = self.svm.predict(X) == 1
        neg = np.logical_not(pos)
        if len(y[pos]) == len(y) or len(y[neg]) == len(y):
            return
        pos_count = np.bincount(y[pos])
        neg_count = np.bincount(y[neg])
        if (len(y[pos]) > 0) and (len(pos_count) > 1) and
        (pos_count[0] / pos_count.sum() > self.max_miss) and (pos_count.min() > 0):
#             print(f"positive region count: {pos_count[1]} true positives out of {len(y[pos])} total")
            self.pos = SVMTree(self.kernel, self.C, self.max_miss, self.boost_strength,
            self.depth+1)
            self.pos.parent = self
```

```
            pos_boost_multiplier = self.boost_strength * (1 + pos_count[1] / pos_count.sum())
            self.pos.fit(X[pos], y[pos], [pos_boost_multiplier if val == 0 else 1 for val in y[pos]])
        if (len(y[neg]) > 0) and (len(neg_count) > 1) and
        (neg_count[1] / neg_count.sum() > self.max_miss) and (neg_count.min() > 0):
#             print(f"negative region count: {neg_count[0]} true negatives out of {len(y[neg])} total")
            self.neg = SVMTree(self.kernel, self.C, self.max_miss, self.boost_strength, self.depth+1)
            self.neg.parent = self
            neg_boost_multiplier = self.boost_strength * (1 + neg_count[0] / neg_count.sum())
            self.neg.fit(X[neg], y[neg], [neg_boost_multiplier if val == 1 else
            1 for val in y[neg]])
        if self.pos is not None and self.neg is not None:
            self.height = max(self.pos.height, self.neg.height) + 1
        elif self.pos is not None:
            self.height = self.pos.height + 1
        elif self.neg is not None:
            self.height = self.neg.height + 1

    def predict(self, X, recurse=True):
        pred = self.svm.predict(X)
        if not recurse:
            return pred
        pos = pred == 1
        neg = np.logical_not(pos)
        if pos.any() and self.pos is not None:
            pred[pos] = self.pos.predict(X[pos, :])
        if neg.any() and self.neg is not None:
            pred[neg] = self.neg.predict(X[neg, :])
        return pred

    def getWeights(self):
        return self.svm.named_steps['model'].intercept_, self.svm.named_steps['model'].coef_

    def plot(self):
        intercept, coef = self.getWeights()
        displacement_vec = -intercept / np.dot(coef[0], coef[0]) * coef[0]
        normal_vec = coef[0]
        # find a projection_vec orthogonal to normal_vec to make the 2nd axis of the plot
        indep_vec = normal_vec.copy()
        indep_vec[normal_vec.nonzero()[0][0]] = 0
        projection_vec = indep_vec - project(indep_vec, normal_vec)

        nfeatures = self.X.shape[1]
        if nfeatures > 2:
            # projection matrix onto space spanned by normal_vec and projection_vec
            P = np.identity(nfeatures)
            for col in range(nfeatures):
                P[:, col] = project(P[:, col], normal_vec) + project(P[:, col], projection_vec)

            # SVD select top 2 influential axes for plotting
            u, s, vh = np.linalg.svd(P)
            axes = np.logical_or(np.arange(nfeatures)==0, np.arange(nfeatures)==1)
            s[np.logical_not(axes)] = 0
            H = np.matmul(np.diag(s), vh)[axes,:]
```

```
            principle_axes = np.matmul(H, np.c_[normal_vec, projection_vec])
            R = rotation_mat2D(principle_axes[:, 0], principle_axes[:, 1])
            H = np.matmul(R, H)
        else:
            H = rotation_mat2D(normal_vec, projection_vec)

        x = np.matmul(self.svm.named_steps['scaler'].transform(self.X), H.T)

        hyperplane = np.matmul(H, displacement_vec)

        fig, ax = plt.subplots()
        ax.scatter(x[:,0], x[:,1], c=self.y, alpha=0.1)
        yellow_patch = mpatches.Patch(color='yellow', label='forward')
        purple_patch = mpatches.Patch(color='purple', label='guard')
        line = ax.axvline(x=hyperplane[0], color='r', label='SVM hyperplane split')
        ax.legend(handles=[yellow_patch, purple_patch, line])
        ax.set_title('Projection plot perpendicular to hyperplane')
        ax.set(xlabel='Span of weight vector')

    def plotBacktrack(self):
        if self.parent is not None:
            self.parent.plotBacktrack()
        self.plot()
```

Fit SVM Tree on 2 features:

```
model_hybrid = SVMTree("linear", 1, 0.1, 1)

start_time = time.perf_counter()

model_hybrid.fit(train_X[:,reb_ast_axes], train_y)

end_time = time.perf_counter()
runtime = end_time - start_time
```

```
print(f"Runtime: {runtime:.6f} seconds")
model_hybrid.height
```

```
y_pred_hybrid = model_hybrid.predict(train_X[:,reb_ast_axes])
print("Hybrid train error: {}".format(
    np.mean(y_pred_hybrid != train_y)
))
```

```
y_pred_hybrid = model_hybrid.predict(test_X[:,reb_ast_axes])
print("Hybrid test error: {}".format(
    np.mean(y_pred_hybrid != test_y)
))
```

```
display = plotDecisionBoundary2D(model_hybrid, train_X[:,reb_ast_axes], train_y)
display.ax_.set(xlabel='REB', ylabel='AST')
plt.show()
```

```
model_hybrid.neg.plotBacktrack()
```

CV Grid Search:

```
def CVgrid(k_folds, model_factory, hyperparameters, X, y, score_fn):
    np.random.seed(42)
    params = np.meshgrid(*hyperparameters.values())
    grid = np.vstack([param.ravel() for param in params]).T
    best_hyp = grid[0]
    best_score = np.inf
    scores = np.zeros(grid.shape[0])
    rand_indices = np.random.permutation(X.shape[0])
    X_folds = np.array_split(X[rand_indices,:], k_folds)
    y_folds = np.array_split(y[rand_indices], k_folds)
    for idx, hyp in enumerate(grid):
        score = []
        for i in range(k_folds):
            val_X = X_folds[i]
            val_y = y_folds[i]
            train_X = np.vstack([X_folds[k] for k in range(k_folds) if k != i])
            train_y = np.hstack([y_folds[k] for k in range(k_folds) if k != i])
            model = model_factory(hyp)
            model.fit(train_X, train_y)
            score.append(score_fn(model.predict(val_X), val_y))
        scores[idx] = np.mean(score)
        print(f'hyp: {hyp}, score: {scores[idx]}')
    best_score = np.max(scores)
    best_hyp = {k:v for k, v in zip(hyperparameters.keys(), grid[np.argmax(scores)])}
    summary = np.c_[grid, scores]
    print(summary)
    print(f'best_hyp: {best_hyp}, best_score: {best_score}')
    return best_hyp, best_score, summary
```

```
start_time = time.perf_counter()
best_hyp, best_score, summary = CVgrid(
    k_folds=5,
    model_factory=lambda hyp: SVMTree('linear', *hyp),
    hyperparameters={'C':[0.1, 1, 10, 100], 'max_miss':[0.001, 0.01, 0.1],
    'boost_strength':[1, 10, 100, 1000]},
    X=train_X,
    y=train_y,
    score_fn=lambda pred, actual: np.mean(pred == actual)
)
end_time = time.perf_counter()
runtime = end_time - start_time
print(f"Runtime: {runtime:.6f} seconds")
```

Fit SVM Tree on full dataset:

```
model_svm_tree = SVMTree(kernel="linear", **best_hyp)

start_time = time.perf_counter()

model_svm_tree.fit(train_X, train_y)

end_time = time.perf_counter()
runtime = end_time - start_time
```

```
print(f"Runtime: {runtime:.6f} seconds")
model_svm_tree.height
```

```
y_pred_svm_tree = model_svm_tree.predict(train_X)
print("SVM Tree train error: {}".format(
    np.mean(y_pred_svm_tree != train_y)
))
```

```
y_pred_svm_tree = model_svm_tree.predict(test_X)
print("SVM Tree test error: {}".format(
    np.mean(y_pred_svm_tree != test_y)
))
```

```
model_svm_tree.neg.pos.plotBacktrack()
```

```
cm = confusion_matrix(test_y, y_pred_svm_tree)
positions = ["Guard", "Forward"]
c_matrix = ConfusionMatrixDisplay(cm, display_labels = positions).plot()
```

Bagging SVM Trees:

```
class BaggedSVMTree:
    def __init__(self, n_models, **kwargs):
        self.models = [SVMTree(**kwargs) for i in range(n_models)]
        self.X = None
        self.y = None

    def fit(self, X, y):
        np.random.seed(42)
        bootstrap_indices = [np.random.randint(X.shape[0], size=X.shape[0]) for i in range(
          len(self.models))]
        self.X = [X[indices, :] for indices in bootstrap_indices]
        self.y = [y[indices] for indices in bootstrap_indices]
        for model, X, y in zip(self.models, self.X, self.y):
            model.fit(X, y)

    def predict(self, X): # majority vote
        return (np.vstack([model.predict(X) for model in self.models]).mean(axis=0) > 0.5).astype(int)
```

```
start_time = time.perf_counter()
best_hyp, best_score, summary = CVgrid(
    k_folds=5,
    model_factory=lambda hyp: BaggedSVMTree(n_models=10, kernel='linear',
    C=hyp[0], max_miss=hyp[1], boost_strength=hyp[2]),
    hyperparameters={
        'C':[0.1, 1, 10, 100, 1000],
        'max_miss':[0.001, 0.01, 0.1, 0.2],
        'boost_strength':[1, 10, 100]
    },
    X=train_X,
    y=train_y,
    score_fn=lambda pred, actual: np.mean(pred == actual)
)
end_time = time.perf_counter()
runtime = end_time - start_time
print(f"Runtime: {runtime:.6f} seconds")
```

Tuning n_models:

```
start_time = time.perf_counter()
best_n, best_score_n, summary_n = CVgrid(
    k_folds=5,
    model_factory=lambda hyp: BaggedSVMTree(n_models=hyp[0], kernel='linear', **best_hyp),
    hyperparameters={'n_models': [10, 20, 40, 80, 160]},
    X=train_X,
    y=train_y,
    score_fn=lambda pred, actual: np.mean(pred == actual)
)
end_time = time.perf_counter()
runtime = end_time - start_time
print(f"Runtime: {runtime:.6f} seconds")
```

```
ns = summary_n[:,0]
val_errors = 1-summary_n[:,1]
plt.plot(ns, val_errors, 'bo-')
for x,y in zip(ns, val_errors):
    label = "{:.4f}".format(y)
    plt.annotate(label, # this is the text
                 (x,y), # these are the coordinates to position the label
                 textcoords="offset points", # how to position the text
                 xytext=(0,10), # distance from text to points (x,y)
                 ha='center') # horizontal alignment can be left, right or center
plt.xlabel("n_models")
plt.ylabel("validation error")
plt.title("Bagged SVM Tree CV summary")
plt.show()
```

```
model_bagged_svm_tree = BaggedSVMTree(n_models=best_n['n_models'], kernel='linear', max_iter=100000, **
```

```
start_time = time.perf_counter()

model_bagged_svm_tree.fit(train_X, train_y)

end_time = time.perf_counter()
runtime = end_time - start_time
```

```
print(f"Runtime: {runtime:.6f} seconds")
[model.height for model in model_bagged_svm_tree.models]
```

```
n_nodes = 0
for m in model_bagged_svm_tree.models:
    n_nodes += m.n_nodes
n_nodes
```

```
np.mean([m.n_nodes for m in model_bagged_svm_tree.models])
```

```
y_pred_bagged_svm_tree = model_bagged_svm_tree.predict(train_X)
print("Bagged SVM Tree train error: {}".format(
    np.mean(y_pred_bagged_svm_tree != train_y)
))
```

```
y_pred_bagged_svm_tree = model_bagged_svm_tree.predict(test_X)
print("Bagged SVM Tree test error: {}".format(
    np.mean(y_pred_bagged_svm_tree != test_y)
))
```

```
cm = confusion_matrix(test_y, y_pred_bagged_svm_tree)
positions = ["Guard", "Forward"]
c_matrix = ConfusionMatrixDisplay(cm, display_labels = positions).plot()
```

```
model_bagged_svm_tree_small = BaggedSVMTree(n_models=best_n['n_models'], kernel='linear', max_iter=1000

start_time = time.perf_counter()

model_bagged_svm_tree_small.fit(train_X, train_y)

end_time = time.perf_counter()
runtime = end_time - start_time
```

```
print(f"Runtime: {runtime:.6f} seconds")
[model.height for model in model_bagged_svm_tree.models]
```

```
n_nodes = 0
for m in model_bagged_svm_tree_small.models:
    n_nodes += m.n_nodes
n_nodes
```

```
np.mean([m.n_nodes for m in model_bagged_svm_tree_small.models])
```

```
y_pred_bagged_svm_tree_small = model_bagged_svm_tree.predict(train_X)
print("Bagged SVM Tree train error: {}".format(
    np.mean(y_pred_bagged_svm_tree_small != train_y)
))
```

```
y_pred_bagged_svm_tree_small = model_bagged_svm_tree.predict(test_X)
print("Bagged SVM Tree test error: {}".format(
    np.mean(y_pred_bagged_svm_tree_small != test_y)
))
```

5 Additional Figures

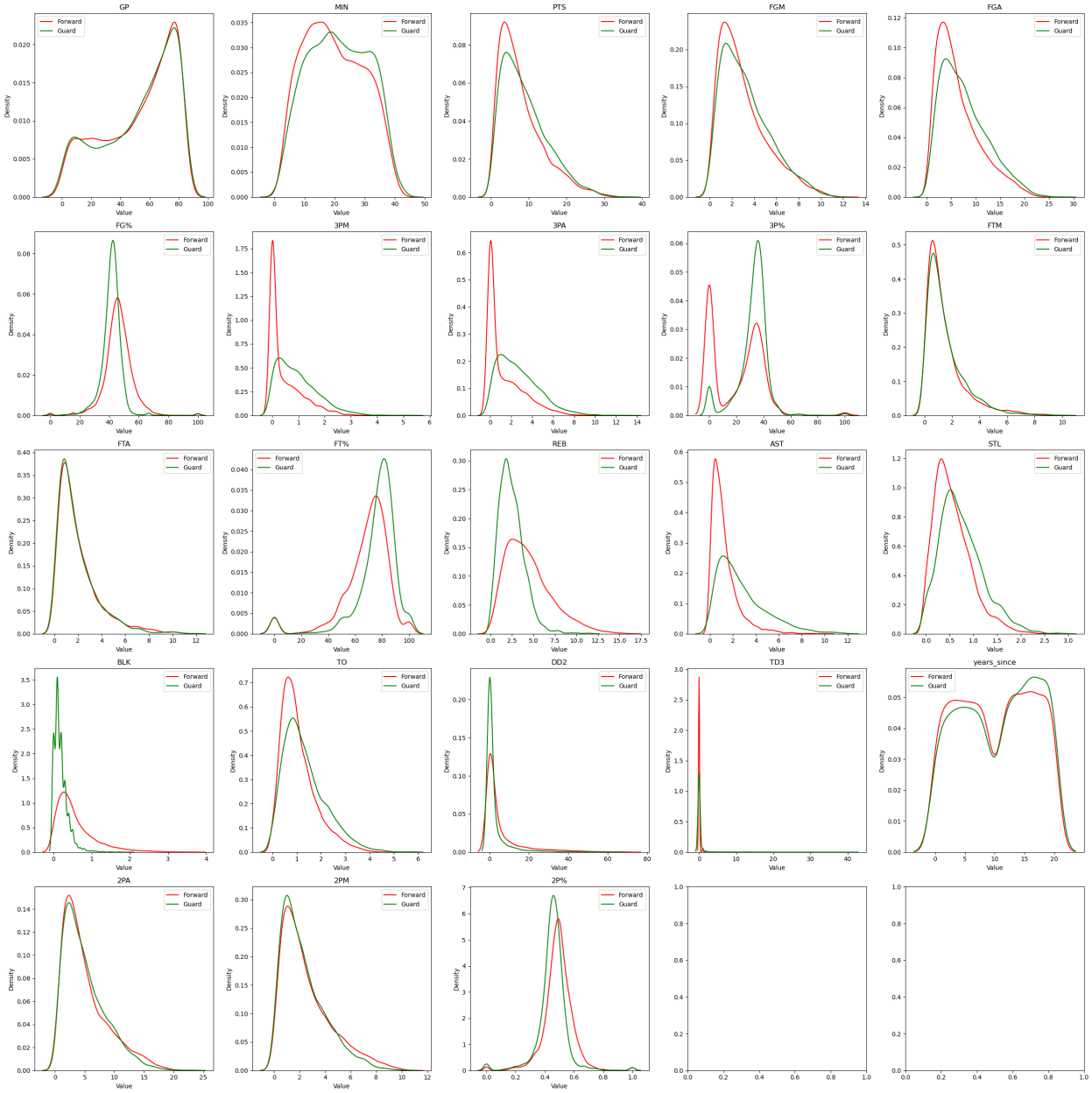

Figure 1: Distribution of Features

SEASON	The NBA regular season the datapoint is from.
NAME	The name of the player.
POS	The position of the player. 0 for Guards and 1 for Forwards.
GP	The number of games a player played in the regular season.
MIN	The number of minutes per game played by a player.
PTS	The number of points scored per game.
FGM	The number of field goals that a player or team has made per game. This includes both 2 pointers and 3 pointers.
FGA	The number of field goals that a player or team has attempted per game. This includes both 2 pointers and 3 pointers.
FG%	The percentage of field goal attempts that a player makes per game. *Formula: (FGM)/(FGA)*
3PM	The number of 3 point field goals that a player has made per game.
3PA	The number of 3 point field goals that a player has attempted per game.
3P%	The percentage of 3 point field goal attempts that a player makes per game.
2PM	The number of 2 point field goals that a player has made per game. *Formula: FGM - 3PM*
2PA	The number of 2 point field goals that a player has attempted per game. *Formula: FGA - 3PA*
2P%	The percentage of 2 point field goal attempts that a player or team makes per game *Formula = (2PM/2PA)*
FTM	The number of free throws that a player has made per game.
FTA	The number of free throws that a player has attempted per game.
FT%	The percentage of free throw attempts that a player has made per game. *Formula = (FTM/FTA)*
REB	A rebound occurs when a player recovers the ball after a missed shot. This statistic is the number of total rebounds a player or team has collected on either offence or defence per game.

Figure 2: Description of Features

All-Around All Star (AAS)	A player belonging to this category is outstanding in many aspects of the game, with statistics that are significantly higher than the league averages in most of the statistical parameters. AAS players can make a difference for their teams in many different ways. They are usually elite scorers, but combine their scoring ability with great passing skills; alternatively, they may have big rebounding numbers or great defensive skills; in some cases, all these characteristics are present at the same time.
Scoring Backcourt (SB)	SB players are characterised by remarkable offensive skills. These players have high PPG statistics, but usually they also have good passing skills, which means high values in AST. They are usually below the average in statistical figures such as TRB and BLK, since they perform better when they play away from the basket.
Scoring Rebounder (SR)	A SR is a player with high-average statistics mainly in PPG and TRB. These players are skilled scorers, both in low-post and facing the basket, with special rebounding abilities. Their size is bigger with respect to SB and they are used to playing closer to the basket. Thanks to these peculiar technical and athletic skills, they take advantage of their position on the field and their size to score from a very short distance.
Paint Protector (PP)	A PP player is usually not a great scorer, with PPG values that are below the averages of the league. They are very good at rebounding and blocking shots, thanks to their size and excellent defensive skills As a result, their values related to TRB, BLK and STL are higher than the other statistical figures. They usually perform a more athletic game and play closer to the basket on both ends of the floor.
Role Player (C)	The RP is usually very good but not excellent in only one statistical category; in some cases, he has numbers that are slightly below the average of the league in all statistical figures. This group includes players who are considered specialists in particular aspects of the game. They can be outside shooters, whose main responsibility is to finalise the actions of other players, or defensive specialists, whose task is to guard the best player of the opponent team. These roles can be crucial in a team, but their importance is usually not reflected in their statistics, which are usually lower than other players.

Figure 3: Player Designations

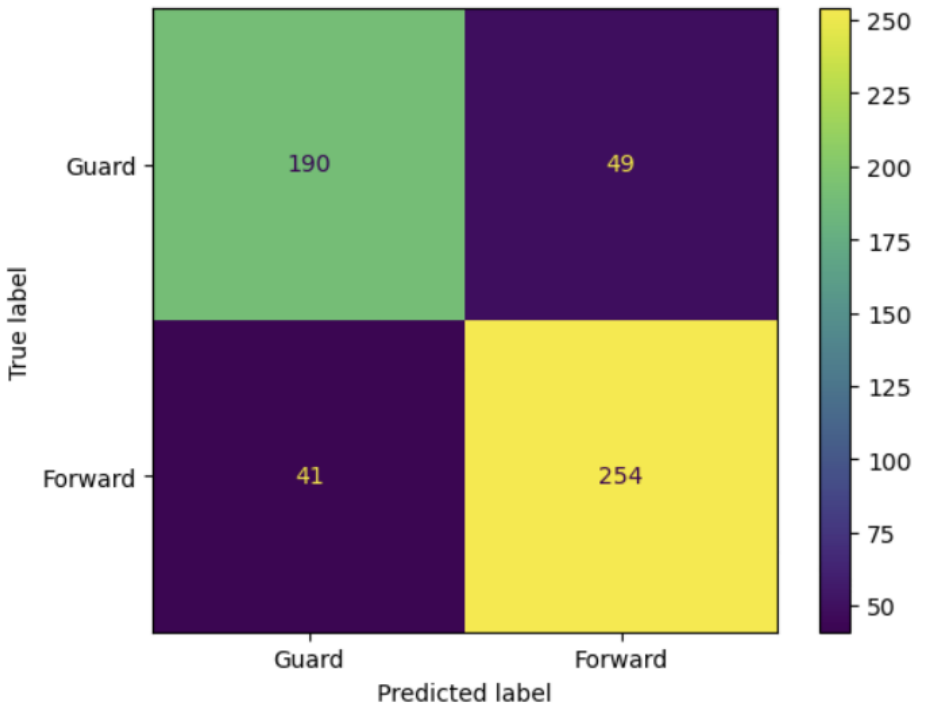

Figure 4: Confusion Matrix for SVM-Tree Bagging Model

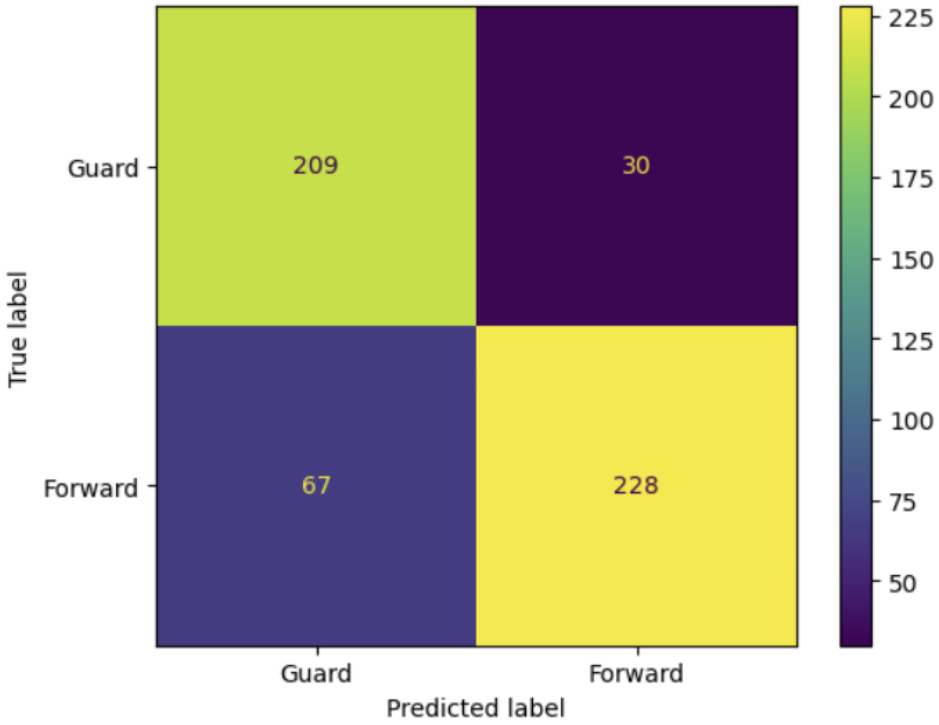

Figure 5: Confusion Matrix for single SVM-Tree Model

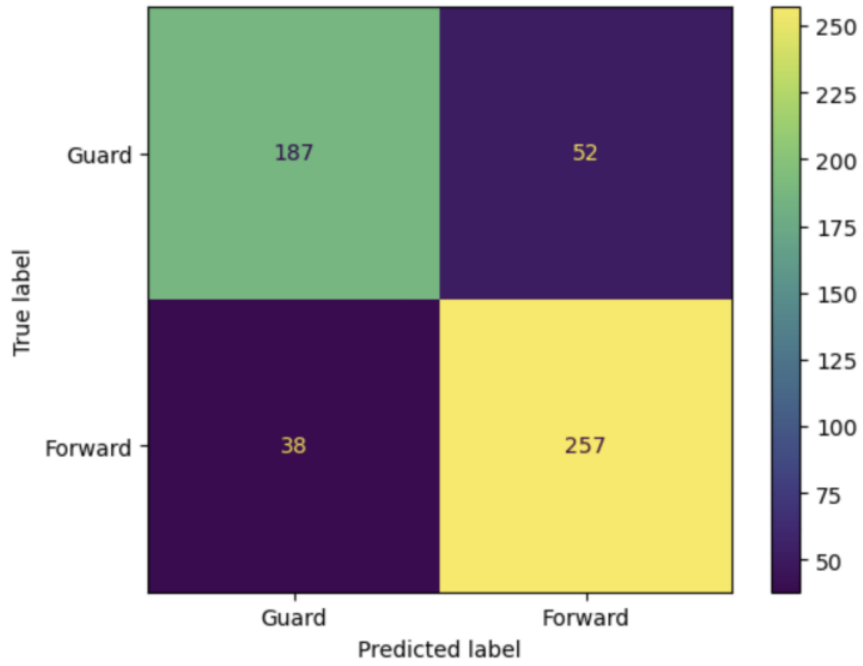

Figure 6: Confusion Matrix for Artificial Neural Network

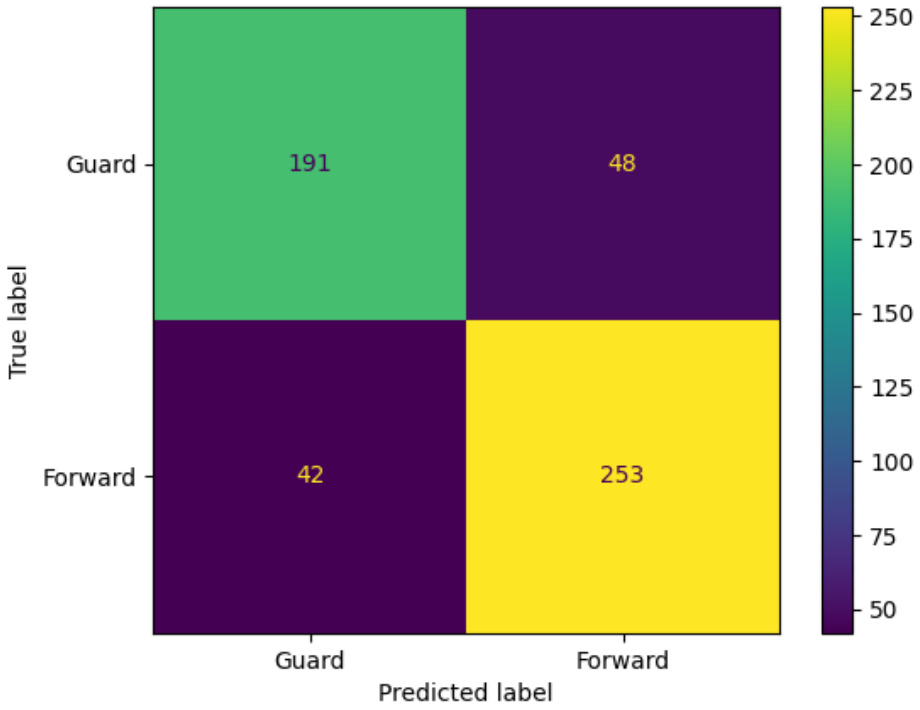

Figure 7: Confusion Matrix for Tree-based Bagging

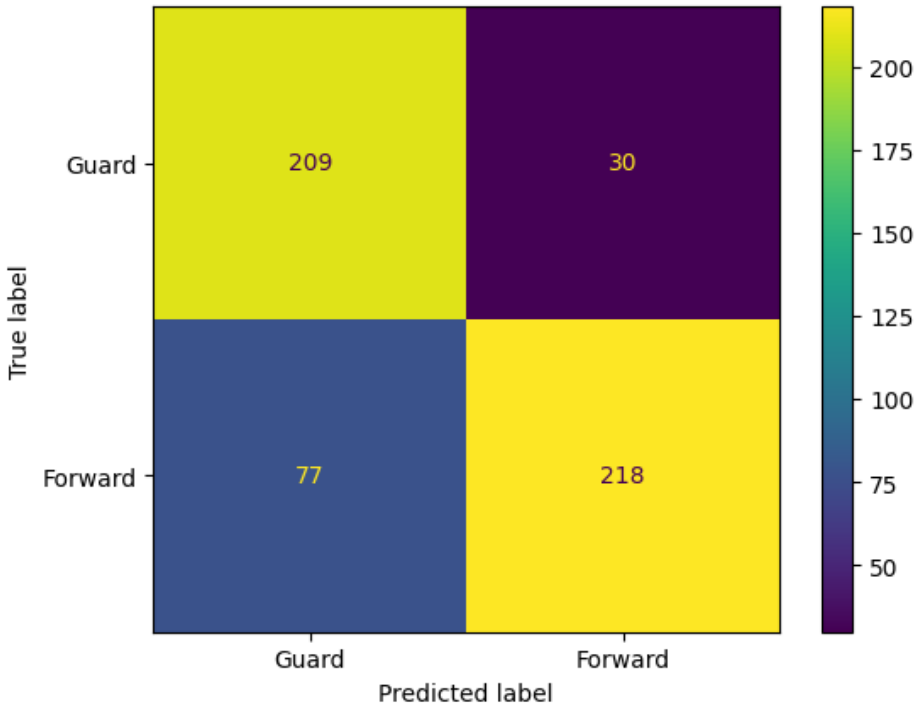

Figure 8: Confusion Matrix for Linear SVM

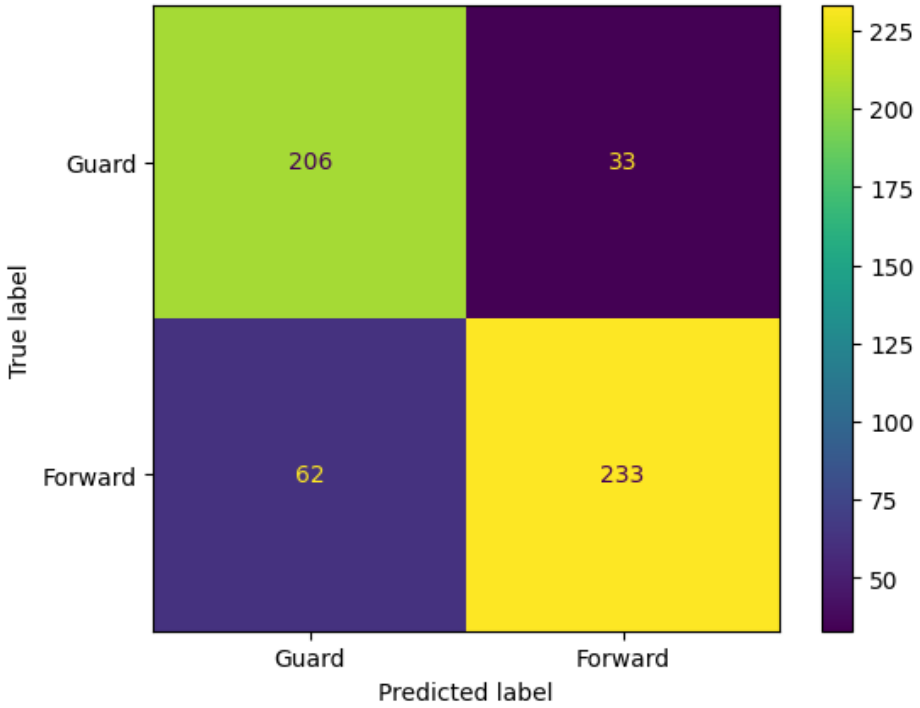

Figure 9: Confusion Matrix for RBF SVM

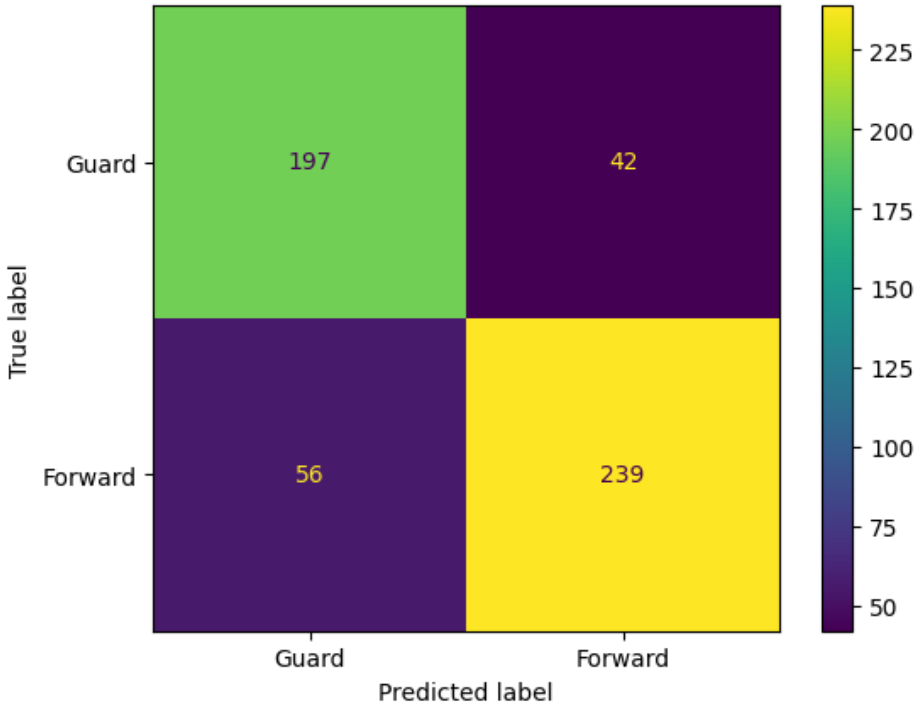

Figure 10: Confusion Matrix for Polynomial SVM