unsupervisedLearningFull

December 9, 2024

[1]:
```
import pandas as pd
import altair as alt
import numpy as np
from sklearn.preprocessing import LabelEncoder
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA
```

1 Project Data and Project Goal

1.1 Data

The data I am using for this project comes from the NFL big data bowl kaggle competetion. There are multiple different CSV files included in the data set. They are #### Games This includes information about game data. Each game has a GameID and information about the teams, time, and scores of the game. #### Player This includes information about player data. Each player has a playerID and information about the players size, background, and position. #### Plays This includes information about each play during a game. It is unique based on gameID and PlayID. It contains a large amount of information about the play both presnap and postsnap. #### PlayerPlay This includes information about each player during a play. It is unique based on gameID, PlayID and playerID. It contains all information about a specific player on a specific play. #### Tracking This includes information about each player every .1 seconds (frames) during a play. It is uniqed based on gameID, PlayID and playerID. It contains the largest amount of information in the dataset including player position, orientation, acceleration and speed in each frame.

1.2 Project Goal

For the big Data bowl this year the goal is too find new metrics about pre-snap data to help predict post snap data. With this project, I want to explore plays that are first down and ten yards to go. Based on presnap features in Plays, I want to use unsupervised ML models to cluster those plays together. I then plan to evaluate all clusters on what makes them similar and also see how

they performed on success metrics like yardsGained on the play and also if the play was a run or pass. Ideally, this will give insights into specific presnap combinations to explore and extrapolate on. I also plan on engineering features to add to the pre snap data to give the model more data to cluster with.

[2]:
```
games = pd.read_csv('games.csv')
playerPlay = pd.read_csv('player_play.csv')
plays = pd.read_csv('plays.csv')
players = pd.read_csv('players.csv')
tracking1 = pd.read_csv('tracking_week_1.csv')
tracking2 = pd.read_csv('tracking_week_2.csv')
tracking3 = pd.read_csv('tracking_week_3.csv')
tracking4 = pd.read_csv('tracking_week_4.csv')
tracking5 = pd.read_csv('tracking_week_5.csv')
tracking6 = pd.read_csv('tracking_week_6.csv')
tracking7 = pd.read_csv('tracking_week_7.csv')
tracking8 = pd.read_csv('tracking_week_8.csv')
tracking9 = pd.read_csv('tracking_week_9.csv')
```

[3]:
```
print(tracking1.head())

#combine all tracking data into one data frame
trackingData = pd.
  ↪concat([tracking1,tracking2,tracking3,tracking4,tracking5,tracking6,tracking7,tracking8,tra
```

```
       gameId  playId    nflId     displayName  frameId    frameType  \
0  2022091200      64  35459.0  Kareem Jackson        1  BEFORE_SNAP
1  2022091200      64  35459.0  Kareem Jackson        2  BEFORE_SNAP
2  2022091200      64  35459.0  Kareem Jackson        3  BEFORE_SNAP
3  2022091200      64  35459.0  Kareem Jackson        4  BEFORE_SNAP
4  2022091200      64  35459.0  Kareem Jackson        5  BEFORE_SNAP

                    time  jerseyNumber club playDirection      x      y     s  \
0  2022-09-13 00:16:03.5          22.0  DEN         right  51.06  28.55  0.72
1  2022-09-13 00:16:03.6          22.0  DEN         right  51.13  28.57  0.71
2  2022-09-13 00:16:03.7          22.0  DEN         right  51.20  28.59  0.69
3  2022-09-13 00:16:03.8          22.0  DEN         right  51.26  28.62  0.67
4  2022-09-13 00:16:03.9          22.0  DEN         right  51.32  28.65  0.65

      a   dis       o    dir                  event
0  0.37  0.07  246.17  68.34  huddle_break_offense
1  0.36  0.07  245.41  71.21                   NaN
2  0.23  0.07  244.45  69.90                   NaN
3  0.22  0.07  244.45  67.98                   NaN
4  0.34  0.07  245.74  62.83                   NaN
```

2 EDA Section

The main data I care about is the data present in the plays dataframe. This data gives a large amount of information about each play. I also plan to use the tracking data and player play data to create other play metrics. The play features I am considering to use in training or evaluation of the models are quarter, down, yardsToGo, yardLine and TeamSide, preSnapVisitorScore, presnapHomeScore, gameClock, possessionTeam, offenseFormation, receiverAlignment, yardsGained, isDropback.
The other features I plan to use outside the play dataframe are motionSinceLineset which is given in the player dataFrame. I also want to calculate offset of the players compared to the ball on the line of scrimmage just pre snap. I will also be changing some of the play features to be more easily used by the models.

I want to filter the play data so that I am only getting first and 10 plays.

[4]:
```
firstAndTenPlays = plays[(plays['down'] == 1) & (plays['yardsToGo'] == 10)]
print(firstAndTenPlays.shape)
```

```
(6408, 50)
```

I also want to discard all plays nullified by penalties.

[5]:
```
cleanedPlays = firstAndTenPlays[firstAndTenPlays['playNullifiedByPenalty'] ==␣
 ↪'N']
print(cleanedPlays.shape)
```

```
(6408, 50)
```

Next, I want to check if their are null values in the features I want to use in the model and I want to remove columns that are not going to be used for the model training or evaluation

[6]:
```
print(cleanedPlays.columns)
cleanedPlays = cleanedPlays.drop(['playDescription', 'targetX',␣
 ↪'targetY','passTippedAtLine','pff_passCoverage', 'pff_manZone'], axis=1)
```

```
Index(['gameId', 'playId', 'playDescription', 'quarter', 'down', 'yardsToGo',
       'possessionTeam', 'defensiveTeam', 'yardlineSide', 'yardlineNumber',
       'gameClock', 'preSnapHomeScore', 'preSnapVisitorScore',
       'playNullifiedByPenalty', 'absoluteYardlineNumber',
       'preSnapHomeTeamWinProbability', 'preSnapVisitorTeamWinProbability',
       'expectedPoints', 'offenseFormation', 'receiverAlignment',
       'playClockAtSnap', 'passResult', 'passLength', 'targetX', 'targetY',
       'playAction', 'dropbackType', 'dropbackDistance', 'passLocationType',
       'timeToThrow', 'timeInTackleBox', 'timeToSack', 'passTippedAtLine',
       'unblockedPressure', 'qbSpike', 'qbKneel', 'qbSneak',
       'rushLocationType', 'penaltyYards', 'prePenaltyYardsGained',
       'yardsGained', 'homeTeamWinProbabilityAdded',
       'visitorTeamWinProbilityAdded', 'expectedPointsAdded', 'isDropback',
       'pff_runConceptPrimary', 'pff_runConceptSecondary', 'pff_runPassOption',
       'pff_passCoverage', 'pff_manZone'],
      dtype='object')
```

Next I want to increase Altairs data point limit to above the number of plays in cleaned plays (6408)

[7]:
```
alt.data_transformers.enable('default', max_rows=7500)
```

[7]:
```
DataTransformerRegistry.enable('default')
```

[8]:
```
#checking for null values in the data
print(cleanedPlays.isnull().sum())
```

```
gameId                                0
playId                                0
quarter                               0
down                                  0
yardsToGo                             0
possessionTeam                        0
defensiveTeam                         0
yardlineSide                         96
yardlineNumber                        0
gameClock                             0
preSnapHomeScore                      0
preSnapVisitorScore                   0
playNullifiedByPenalty                0
absoluteYardlineNumber                0
preSnapHomeTeamWinProbability         0
preSnapVisitorTeamWinProbability      0
expectedPoints                        0
offenseFormation                    114
receiverAlignment                   114
playClockAtSnap                       1
passResult                         3231
passLength                         3482
playAction                            0
dropbackType                       3050
dropbackDistance                   3140
passLocationType                   3346
timeToThrow                        3494
timeInTackleBox                    3432
timeToSack                         6260
unblockedPressure                  3235
qbSpike                            3332
qbKneel                               0
qbSneak                            3076
rushLocationType                   3076
penaltyYards                       6288
prePenaltyYardsGained                 0
yardsGained                           0
homeTeamWinProbabilityAdded           0
```

```
visitorTeamWinProbilityAdded          0
expectedPointsAdded                   0
isDropback                            0
pff_runConceptPrimary              1782
pff_runConceptSecondary            4934
pff_runPassOption                     0
dtype: int64
```

yardLineSide is okay since those values indicated the 50 yard line (no teams sideline). I do want to drop null values from offense formation and receiver alignment.

[9]:
```
cleanedPlays = cleanedPlays.dropna(subset=['receiverAlignment',␣
↪'offenseFormation'])
```

Explore features I want to include in the model

[10]:
```
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('quarter:N',title='Quarter'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=500,
    height=300
)
```

[10]:
```
alt.Chart(…)
```

It looks like there is an even distribution of plays throughout the game with a few overtime plays mixed in.

[11]:
```
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('offenseFormation:N',title='Offense Formation'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=500,
    height=300
)
```

[11]:
```
alt.Chart(…)
```

There is a variety of different formations with some more popular than others. I will standardize and encode this categorical data later on.

[12]:
```
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('receiverAlignment:N',title='Receiver Alignment'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=700,
    height=300
)
```

[12]: alt.Chart(…)

Similar to the offensive formation data, I plan to standardize and encode this data later on so that it can be better used in the KMeans model and not overly influence the clusters.

[13]:
```
#exploring potential feature to assess clusters on
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('yardsGained:N',title='Yards Gained'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=700,
    height=300
)
```

[13]: alt.Chart(…)

[14]:
```
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('possessionTeam:N',title='Possession Team'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=700,
    height=300
)
```

[14]: alt.Chart(…)

I am not going to use possession teams since I want to focus more on the formation than specific players on specific teams. The team data is also categorical which does not work as well with KMeans and hierarchal clustering. Next I want to transform absoluteYardline into yardsToGoal. As seen in the graph below the absolute yardline is given but this does not indicate yards from the endzone.

[15]:
```
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('absoluteYardlineNumber:N',title='Absolute Yardline Number'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=1000,
    height=300
)
```

[15]: alt.Chart(…)

I thought absolute yardline would give yardsToEndzone for possession team, but it does not. I will use the features below to create a yardsToEndzone feature.

[16]:
```
print(cleanedPlays['yardlineSide'].unique(), cleanedPlays['yardlineNumber'].
↪unique())
```

```
['CIN' 'IND' 'GB' 'LAC' 'PHI' 'CLE' 'PIT' 'MIA' 'DEN' 'CAR' 'DET' 'ATL'
 'MIN' 'HOU' 'CHI' 'LV' 'ARI' 'KC' 'NYJ' 'NO' 'NYG' 'BUF' 'WAS' 'SF' 'NE'
```

```
 'SEA' 'BAL' 'TEN' 'TB' 'DAL' 'LA' nan 'JAX'] [21  8 40 25 42 47 34 36 31 29 44
23 49 38 13 41  9 19 11 10 17 46 20 28
 35 48 37 22 14 12 30 27 43 18 32 39 45 26 24 15 50  7 16 33  6  2  1  5
  3  4]
```

If the possession team is on their side of the field then the yardlineSide value will equal possession team and yards to endzone would be 100 - yardlineNumber, otherwise yards to endzone would just equal yardlineNumber.

[17]:
```
cleanedPlays['yardsToEndzone'] = cleanedPlays.apply(
    lambda row: 100 - row['yardlineNumber'] if row['possessionTeam'] ==␣
 ↪row['yardlineSide'] else row['yardlineNumber'],
    axis=1
)
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X('yardsToEndzone:N',title='Yards to Endzone'),
    y = alt.Y('count():Q',title='Count')
).properties(
    width=1000,
    height=300
)
```

[17]:
```
alt.Chart(…)
```

I want to incorporate a few other presnap features to the cleanPlays dataframe. Incorporate motionSinceLinest and shiftSinceLineset as well as Y offset.

[18]:
```
class TrackingDataProcessor:
    """A class to process and analyze play and game data."""

    def __init__(self, data, plays):
        #data frame of tracking data
        self.data = data
        #data frame of plays
        self.plays = plays

    def get_current_play_data(self,gameId,playId):
        """
        This gets the tracking data for a play in plays
        Returns:
        Dataframe of tracking data that has specific gameId and playId for a␣
 ↪play
        """
        return self.data[(self.data['playId'] == playId) & (self.data['gameId']␣
 ↪== gameId)]

    def get_frame_id(self, currentPlayData):
        """
```

```
        This gets the frameId of a play just before the ball is snapped
        ****neeed to figure out case where penalty is called pre snap*****
        Returns:
        Int: the frame Id
        """
        firstPlayer = currentPlayData['nflId'].iloc[0]
        firstPlayerData = currentPlayData[currentPlayData['nflId'] ==␣
↪firstPlayer]
        lastPresnapIndex = firstPlayerData[firstPlayerData['frameType'] ==␣
↪'SNAP']
        frameId = lastPresnapIndex["frameId"].iloc[0] - 1
        return frameId

    def get_ball_position(self,currentPlayData):
        """
        This gets position of ball on each play
        Returns:
        Tuple of X and Y coordinates
        """
        ballIndex = currentPlayData[currentPlayData["displayName"] ==␣
↪'football']
        ballPosY = ballIndex["y"].iloc[1]
        ballPosX = ballIndex['x'].iloc[1]
        return (ballPosY,ballPosX)


    def get_offset(self, currentPlayFrameData, ballPosition, possessionTeam,␣
↪axis):
        """
        This computes y distance offset from the balls spot for defense and␣
↪offense players

        Returns:
        float: A float that is the difference between the current plays offense␣
↪and defense offset
        """
        offenseOffset = 0
        defenseOffset = 0
        for playerRow in currentPlayFrameData.itertuples():
            if axis == 'x':
                playerPos = playerRow.x
            else:
                playerPos = playerRow.y
            disToBall = ballPosition - playerPos
            if playerRow.club == possessionTeam:
                offenseOffset += disToBall
```

```
            else:
                defenseOffset += disToBall
            offenseOffset = offenseOffset
            defenseOffset = defenseOffset
        return np.abs(offenseOffset - defenseOffset) / 11

    def get_euclidean_offset(self, x, y):
        """
        Uses get_x_offset and get_y_offset to calculate euclidean distance offset for each play

        Returns:
        float: the offset for the play based on x and y offsets
        """
        return np.sqrt(x**2 + y**2)

    def calculate_offsets_at_snap(self):
        """
        Loops through all unique plays and calculates offsets for offense and defense for all plays

        Returns:
        Tuple(yOffset, xOffset, euclideanOffset): arrays of all offsets for each play
        """
        yOffset, xOffset, euclideanOffset = [], [], []
        for row in self.plays.itertuples():
            currentPlayData = self.get_current_play_data(row.gameId, row.playId)
            frameId = self.get_frame_id(currentPlayData)
            #get tracking data for specific frameId
            currentPlayFrameData = currentPlayData[currentPlayData['frameId'] == frameId]
            currentPlayFrameData = currentPlayFrameData.dropna(subset=['nflId'])
            #find possession team for the play
            possessionTeam = row.possessionTeam
            ballPosX, ballPosY = self.get_ball_position(currentPlayData)
            playYOffset = self.get_offset(currentPlayFrameData,ballPosY,possessionTeam,"y")
            playXOffset = self.get_offset(currentPlayFrameData,ballPosX,possessionTeam,"x")
            yOffset.append(playYOffset)
            xOffset.append(playXOffset)
            if playYOffset != None:
                euclideanOffset.append(self.get_euclidean_offset(playYOffset,playXOffset))
            else:
```

```
                euclideanOffset.append(None)
        return (yOffset,xOffset,euclideanOffset)
```

[19]:
```
#using Processor class to create three features for my data.
offsetsCalculator = TrackingDataProcessor(trackingData, cleanedPlays)
yOffsetAtSnap, xOffsetAtSnap, euclideanOffsetAtSnap = offsetsCalculator.
 ↪calculate_offsets_at_snap()
cleanedPlays['yOffsetAtSnap'] = yOffsetAtSnap
cleanedPlays['xOffsetAtSnap'] = xOffsetAtSnap
cleanedPlays['euclideanOffsetAtSnap'] = euclideanOffsetAtSnap
```

[20]:
```
#using player play data frame to create two new features in cleaned plays␣
 ↪dataframe
playerPlayMotion = playerPlay.merge(cleanedPlays[['gameId','playId']],␣
 ↪on=['gameId', 'playId'])
motion_status = playerPlayMotion.groupby('playId')['motionSinceLineset'].any().
 ↪reset_index()
cleanedPlays = cleanedPlays.merge(motion_status, on='playId', how='left')
print(cleanedPlays['motionSinceLineset'])
playerPlayMotion = playerPlay.merge(cleanedPlays[['gameId','playId']],␣
 ↪on=['gameId', 'playId'])
shift_status = playerPlayMotion.groupby('playId')['shiftSinceLineset'].any().
 ↪reset_index()
cleanedPlays = cleanedPlays.merge(shift_status, on='playId', how='left')
print(cleanedPlays['shiftSinceLineset'].head())
```

```
0        True
1       False
2       False
3        True
4        True
        …
6289     True
6290    False
6291    False
6292     True
6293    False
Name: motionSinceLineset, Length: 6294, dtype: bool
0       False
1       False
2       False
3       False
4       False
        …
```

```
6289     True
6290    False
6291    False
6292    False
6293     True
Name: shiftSinceLineset, Length: 6294, dtype: bool
```

Added yOffset, xOffset, euclideanOffset, motionSinceLineset and shiftSinceLineset to cleanedPlays. Going through 6000 plays and evaluating tracking data for each play takes minutes to complete, but fortunately, I only need to run this once. Lastly, I want to adjust game clock to run from 60 mins down to 0 instead of 15 min quarters. This will combine gameClock and quarter features.

[21]:
```
#combine gameClock and quarter features
print(type(cleanedPlays['gameClock'].iloc[0]), type(cleanedPlays['quarter'].
 ↪iloc[0]))
#need to convert string to int for gameClock and also use quarter
cleanedPlays['secondsLeft'] = cleanedPlays.apply(lambda row:␣
 ↪int(row['gameClock'].split(":")[0]) * 60 + int(row['gameClock'].split(":
 ↪")[1]) + (4 - row['quarter']) * 15 * 60, axis=1)
#I also want to drop plays that have less than 120 seconds left. I think these␣
 ↪are usually hurry up plays that are unique to end game situations
cleanedPlays = cleanedPlays[cleanedPlays["secondsLeft"] >= 120]
```

```
<class 'str'> <class 'numpy.int64'>
```

Also want to add feature score difference that calcuates the possession teams score difference. Positive means they are up and negative means they are down. Reducing formations to just shotgun, singleback and other so they do not skew that clustering algorithm. Lastly turning reciever alignment into reciever differential e.g. 1x3 would be 2 and 2x2 would be 0. Just taking the absolute value of the difference between the two sides.

[22]:
```
#also want to combine home team presnap score and away team presnap score to␣
 ↪just home team - away team score
cleanedPlays["scoreDifference"] = np.where(cleanedPlays['preSnapHomeScore'] ==␣
 ↪cleanedPlays['possessionTeam'],
                          cleanedPlays['preSnapHomeScore'] -␣
 ↪cleanedPlays['preSnapVisitorScore'],
                          cleanedPlays['preSnapVisitorScore'] -␣
 ↪cleanedPlays['preSnapHomeScore'])
```

[24]:
```
#Decided against simplifying these features since they do bring important␣
 ↪information to the model.

# cleanedPlays["reducedFormations"] = cleanedPlays['offenseFormation'].apply(
#     lambda y: y if y in ['SHOTGUN', 'SINGLEBACK'] else 'other')
# print(cleanedPlays["reducedFormations"])
```

```
# cleanedPlays['receiverDifferential'] = cleanedPlays['receiverAlignment'].
 ↪apply(
#     lambda x: np.abs(int(x.split("x")[0]) - int(x.split("x")[1])))
# print(cleanedPlays["receiverDifferential"])
```

Create Box plots of relevant data, and also look for outliers. I think for the nfl data, I do not want to remove outliers since those are successful or unsuccessful plays for a reason and I want that data to be involved in the model.

[25]:
```
alt.Chart(cleanedPlays).mark_boxplot().encode(
    x='offenseFormation:N',
    y='yardsGained:Q',
    color='offenseFormation:N'
).properties(
    title="Box Plot of Yards Gained by Offense Formation",
    width=600,
    height=300
)
```

[25]: alt.Chart(…)

[26]:
```
alt.Chart(cleanedPlays).mark_boxplot().encode(
    x= alt.X('quarter:N'),
    y= alt.Y('yardsGained:Q'),
    color='quarter:N'
).properties(
    title="Box Plot of yards gained in each quarter",
    width=400,
    height=300
)
```

[26]: alt.Chart(…)

[27]:
```
alt.Chart(cleanedPlays).mark_boxplot().encode(
    x='receiverAlignment:N',
    y='yardsGained:Q',
    color='receiverAlignment:N'
).properties(
    title="Box Plot for Yards Gained Based on Receiver Alignment",
    width=700,
    height=300
)
```

[27]: alt.Chart(…)

Check for covariance and collinearity between features Features I want to use: ('secondsLeft', 'scoreDifference', 'offenseFormation', 'receiverAlignment', 'yardsToEndzone', 'yOffsetAtSnap', 'xOffsetAtSnap', 'euclideanOffsetAtSnap', 'motionSinceLineset', 'shiftSinceLineset') 10 features to evaluate 2 features are categorical 2 features are booleans 6 features are numeric need to

convert categorical features via label encoder to see if there is collinearity and covariance

[28]:
```
cleanedPlayFeatures = cleanedPlays[['secondsLeft', 'scoreDifference',␣
 ↪'xOffsetAtSnap', 'offenseFormation', 'receiverAlignment', 'yardsToEndzone',␣
 ↪'yOffsetAtSnap', 'euclideanOffsetAtSnap', 'motionSinceLineset',␣
 ↪'shiftSinceLineset']]
cleanedPlayFeatures.info()
```

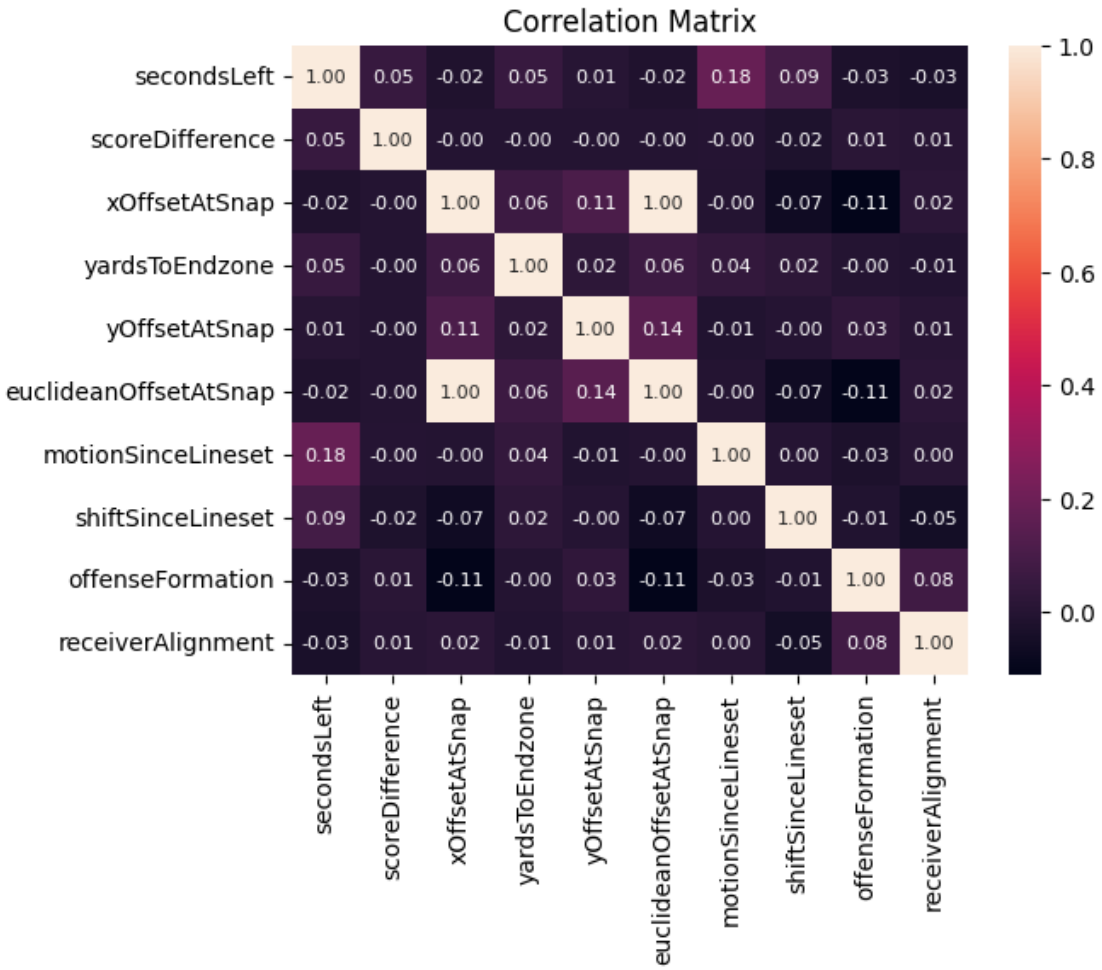
```
<class 'pandas.core.frame.DataFrame'>
Index: 6036 entries, 0 to 6293
Data columns (total 10 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   secondsLeft            6036 non-null   int64
 1   scoreDifference        6036 non-null   int64
 2   xOffsetAtSnap          6036 non-null   float64
 3   offenseFormation       6036 non-null   object
 4   receiverAlignment      6036 non-null   object
 5   yardsToEndzone         6036 non-null   int64
 6   yOffsetAtSnap          6036 non-null   float64
 7   euclideanOffsetAtSnap  6036 non-null   float64
 8   motionSinceLineset     6036 non-null   bool
 9   shiftSinceLineset      6036 non-null   bool
dtypes: bool(2), float64(3), int64(3), object(2)
memory usage: 436.2+ KB
```

[29]:
```
catFeatures = cleanedPlayFeatures.select_dtypes(include=['object'])
numericFeatures = cleanedPlayFeatures.select_dtypes(include=['int','float'])
boolFeatures = cleanedPlayFeatures.select_dtypes(include=['bool']).astype(int)

encodedFeatures = catFeatures.apply(LabelEncoder().fit_transform)

allFeatures = pd.concat([numericFeatures, boolFeatures, encodedFeatures],␣
 ↪axis=1)

sns.heatmap(allFeatures.corr(), annot=True, fmt=".2f",annot_kws={"size": 8},)
plt.title("Correlation Matrix")
plt.show()
```



Interestingly, there is only one strong feature correlations in the data which makes sense since xOffsetAtSnap is the major variable in determining Euclidean Distance at Snap

[30]:
```
#remove XOffset
allFeatures = allFeatures.drop(columns=["xOffsetAtSnap"])

#standardize features
scaledFeatures = StandardScaler().fit_transform(allFeatures)

# Compute VIF
vifData = pd.DataFrame()
vifData['feature'] = allFeatures.columns
vifData['score'] = [variance_inflation_factor(scaledFeatures, i) for i in␣
 ↪range(scaledFeatures.shape[1])]
print(vifData)
```

```
                feature     score
```

```
0            secondsLeft  1.048322
1        scoreDifference  1.003771
2         yardsToEndzone  1.008265
3          yOffsetAtSnap  1.024130
4  euclideanOffsetAtSnap  1.045207
5     motionSinceLineset  1.035586
6      shiftSinceLineset  1.015632
7       offenseFormation  1.022500
8      receiverAlignment  1.010086
```

[31]:
```
# Drop Euclidean distance since it is a feature calculated with xOffset and␣
 ↪yOffset, also dropping xOffsetAtSnap since I don't care too much about this␣
 ↪variable either.
# The Xoffset should only be large on long hail mary type plays. This could␣
 ↪skew the data.
allFeatures = allFeatures.drop(columns=['euclideanOffsetAtSnap'], axis=1)
#rerun code
scaledFeatures = StandardScaler().fit_transform(allFeatures)

# Compute VIF
vifData = pd.DataFrame()
vifData['feature'] = allFeatures.columns
vifData['score'] = [variance_inflation_factor(scaledFeatures, i) for i in␣
 ↪range(scaledFeatures.shape[1])]
print(vifData)
```

```
             feature     score
0        secondsLeft  1.047956
1    scoreDifference  1.003765
2     yardsToEndzone  1.004198
3      yOffsetAtSnap  1.001720
4  motionSinceLineset  1.035565
5   shiftSinceLineset  1.011029
6    offenseFormation  1.008118
7   receiverAlignment  1.009534
```

Much better, this indicates that no variable can be linearly predicted by a combination of other variables. The higher the VIF score the more multicollinearity a feature has and this can interfere with the model. Since these features are multicollinear with other variables they may be more heavily clustered together than what would be desired in a clustering model. It also reduces the dimensions of the data making the model focus on what is important.

[63]:
```
allFeatures.to_csv("allFeatures.csv", index=False)
cleanedPlays.to_csv("cleanedPlays.csv",index=False)
```

[ ]:
```

```

3 Model

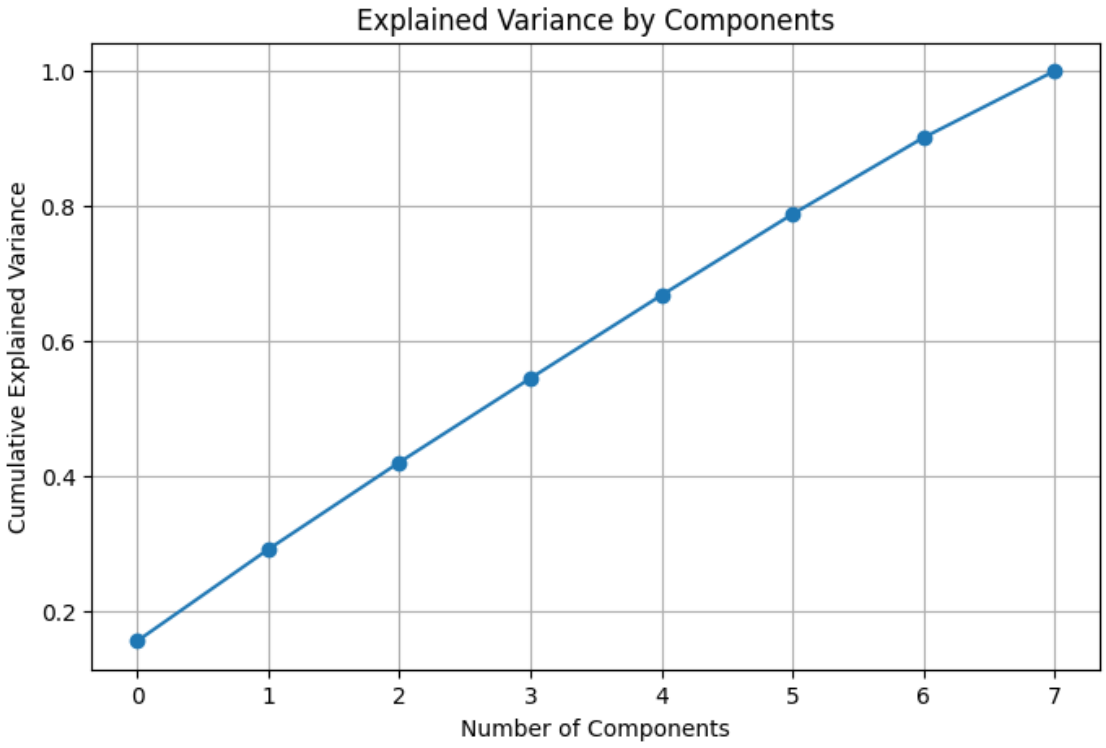
KMeans I think PCA would not be very helpful with this dataset because they all have very low VIF scores (Score on multicollinearity) and the number of features is not large.

I am planning on using KMeans and hierarchical Clustering to find clusters in the data and see what feature combinations fit together. Then I am going to evaluate them on yardsGained and other post snap metrics to see if the groupings reveal any patterns that the presnap data clusters could predict.

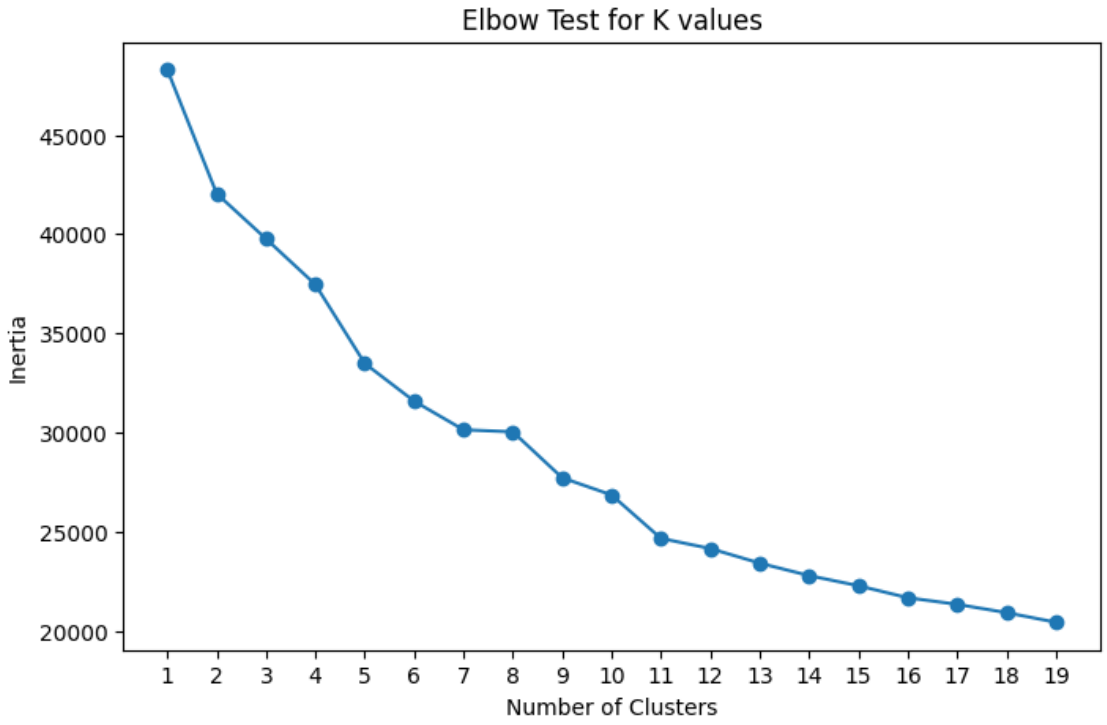
I made sure to scale the features for kmeans so distances are not affected by the feature units.

[26]:
```
#PCA on scaled features
pca = PCA()
pca.fit(scaledFeatures)  # Use your scaled data

# Plot cumulative explained variance
plt.figure(figsize=(8, 5))
plt.plot(np.cumsum(pca.explained_variance_ratio_), marker='o')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance by Components')
plt.grid()
plt.show()
```



[27]:
```
#inertia is sum of squared distances for points around there centroid, once the
 ↪interia starts to decrease less is the elbow point.  This point is where k
 ↪should be selected based on this method.
inertias = []
kRange = range(1,20)
for k in kRange:
    kmeansModel = KMeans(n_clusters=k, random_state=0)
    kmeansModel.fit(scaledFeatures)
    inertias.append(kmeansModel.inertia_)
plt.figure(figsize=(8, 5))
plt.plot(kRange, inertias, marker='o')
plt.title('Elbow Test for K values')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.xticks(kRange)
plt.show()
```

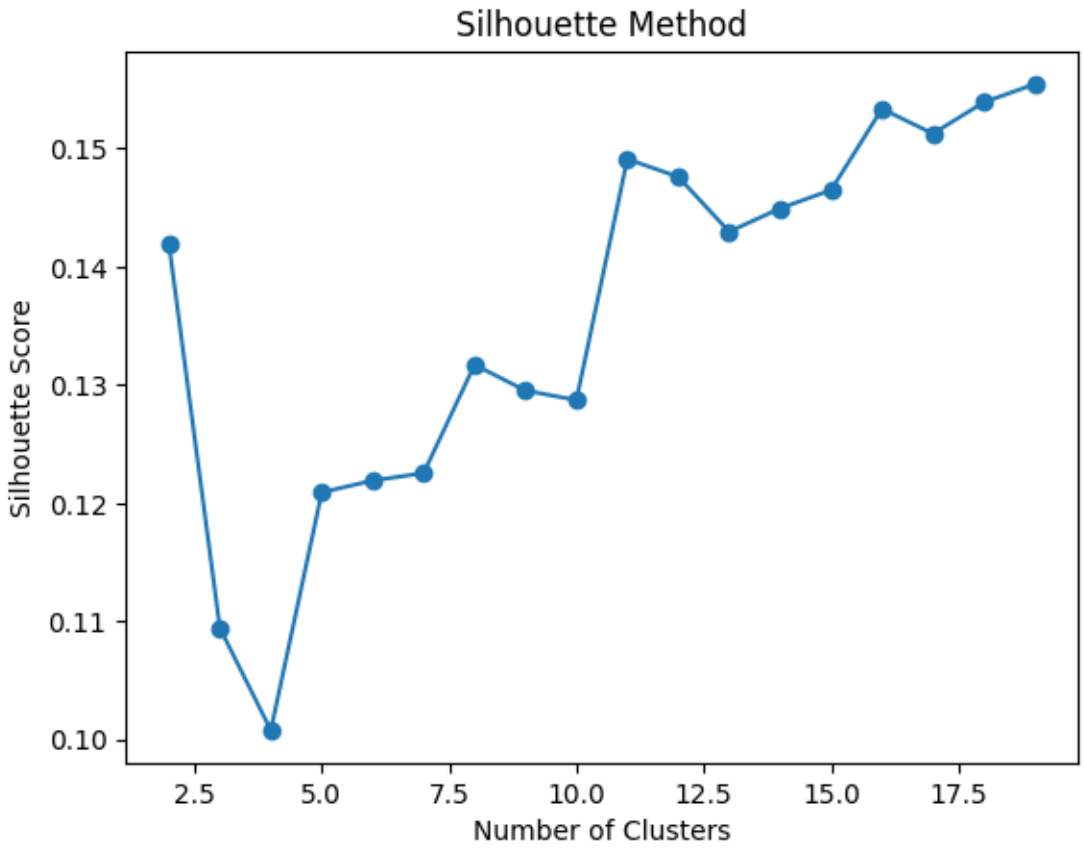


It looks like valid Elbow k values could be 5 and 7. Both of these values have lower decreases in interia after them compared to before them. I'll first use 5 since the drop off through the first five points is larger than the next 15. Additionally, I want to evaluate for the right k value using the silhouette test.

PCA Was not needed as you can see in the PCA plot. Every feature is nearly equally important. Additionally, there is not an overwhelming number of features for the Kmeans model so we do not

have to worry about the curse of dimensionality.

[28]:
```
scores = []
jRange = range(2,20)
for k in jRange:
    kMeansModel = KMeans(n_clusters=k, random_state=0)
    kMeansModel.fit(scaledFeatures)
    scores.append(silhouette_score(scaledFeatures, kMeansModel.labels_))
plt.plot(jRange, scores, marker='o')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Method')
plt.show()
```



A silhouette Score is a score assigned each data point in the model. It indicates how certain a point should be apart of a cluster compared to its closest neighbor. It ranges from 1 (very certain it should be apart of the cluster to -1 probably should be apart of a different cluster). These scores are near zero so are apoart of a cluster but could reasonably move to the neighbor. Most notably, there is a jump at k = 5 and 7 which is the same points we distinguished in the elbow graph. This gives me more certainty that either would be a good number of clusters for the KMeans model.

[29]:
```
#now train KMeans with k = 5
kMeansModel = KMeans(n_clusters=5, random_state=0)
kMeansModel.fit(scaledFeatures)
cleanedPlays['kMeansLabel'] = kMeansModel.labels_
print(cleanedPlays['kMeansLabel'].value_counts())
```

```
kMeansLabel
3    1666
1    1443
0    1064
2     982
4     881
Name: count, dtype: int64
```

[30]:
```
alt.Chart(cleanedPlays).mark_boxplot().encode(
    x = alt.X("kMeansLabel:N"),
    y = alt.Y("yardsGained:Q"),
    color = "kMeansLabel:N"
).properties(
    title = "label compared with yards gained",
    width = 700,
    height = 400)
```

[30]: alt.Chart(…)

[31]:
```
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X("kMeansLabel:N"),
    y = alt.Y("count():Q"),
    color = "isDropback",
    tooltip=['kMeansLabel', 'isDropback', 'count()'],
    xOffset='isDropback:N'
).properties(
    title = "Pass/Run play",
    height=300,
    width=600)
```

[31]: alt.Chart(…)

Interestingly, The model grouped the data with group 1 having a higher average yards per play while also having a higher than average dropback rate. This is a cluster I would want to explore in more detail to see how those plays were grouped. But first, I am going to train a hierarchical clustering model on the same data to see if results are similar or different.

3.1 Model 2

I want to use the same scaled data as the KMeans model. The KMeans model also helped me find a good clustering number, so I plan to use 5 clusters for my heirarchical clustering model. For parameters, I want to test ward linkage and average linkage for the linkage type. This is because I included outliers in the data and using single or complete linkage would create too much emphasis

on the outliers in the data. For affinity, I am curious about using euclidean, l1 and manhattan. These two distances are magnitude difference based and I think this will work well since a majority of my data is numeric. Since there are only four combinations of these hyperparameters (only three valid ones), I am going to assess each of these outcomes graphically compared to yardsGained and isDropback.

[32]:
```
linkages = ['ward','average']
affinities = ['euclidean','manhattan','l1']
labels = ['heirLabel1','heirLabel2','heirLabel3','heirLabel4']
indexer = 0
saveCombinations = []
#looping through all hyperparameter options
for link in linkages:
    for affin in affinities:
        #ward can only be used with euclidean
        if ((link != 'ward' or affin != 'manhattan') and (link != 'ward' or␣
 ↪affin != 'l1')):
            #training model with scaledFeatures
            hierModel = AgglomerativeClustering(n_clusters=5, metric=affin,␣
 ↪linkage=link)
            hierModel.fit(scaledFeatures)
            cleanedPlays[labels[indexer]] = hierModel.labels_
            saveCombinations.append([link,affin,labels[indexer]])
            indexer += 1
```

[33]:
```
print(saveCombinations)
```

```
[['ward', 'euclidean', 'heirLabel1'], ['average', 'euclidean', 'heirLabel2'],
['average', 'manhattan', 'heirLabel3'], ['average', 'l1', 'heirLabel4']]
```

[34]:
```
#display all of the combinations compared with yards Gained
charts = []
for label in labels:
    charts.append(alt.Chart(cleanedPlays).mark_boxplot().encode(
            x = alt.X(label + ":N"),
            y = alt.Y("yardsGained:Q"),
            color = label + ":N"
        ).properties(
            title = "label compared with yards gained",
            width = 700,
            height = 400))
    print(cleanedPlays[label].value_counts())
charts[0] & charts[1] & charts[2] & charts[3]
```

```
heirLabel1
```

```
0    2067
1    1771
3    1289
2     618
4     291
Name: count, dtype: int64
heirLabel2
0    5617
4     291
1     120
3       5
2       3
Name: count, dtype: int64
heirLabel3
0    5727
3     291
4       8
1       7
2       3
Name: count, dtype: int64
heirLabel4
0    5727
3     291
4       8
1       7
2       3
Name: count, dtype: int64
```

[34]:
```
alt.VConcatChart(…)
```

So there is a difference from hierarchical clustering using ward/euclidean vs other features. It seems like the most even clusters are with ward/euclidean and the other hyper parameters skew more toward a few small clusters and one very large cluster.

[35]:
```
#investigate outlier
outliers = cleanedPlays[cleanedPlays['heirLabel2'] == 2]
print(outliers[['secondsLeft','yardsToEndzone', 'scoreDifference',␣
 ↪'yOffsetAtSnap', 'receiverAlignment', 'offenseFormation',␣
 ↪'motionSinceLineset', 'shiftSinceLineset']])
```

```
      secondsLeft  yardsToEndzone  scoreDifference  yOffsetAtSnap  \
4390          446              56              -24       0.332727
5079         1317              10              -20       0.920000
5578         1653              49              -17       0.070000

     receiverAlignment offenseFormation  motionSinceLineset  shiftSinceLineset
4390               1x1          WILDCAT               False               True
5079               1x1          WILDCAT               False               True
5578               1x1          WILDCAT                True               True
```

Next need to evaluate interesting model clusters

[36]:
```
charts = []
for label in labels:
    charts.append(alt.Chart(cleanedPlays).mark_bar().encode(
            x = alt.X(label + ":N"),
            y = alt.Y("count():Q"),
            color = "isDropback",
            tooltip=[label, 'isDropback', 'count()'],
            xOffset='isDropback:N'
        ).properties(
            title = "label compared with isDropback (meaning true = pass, false= run)",
            width = 700,
            height = 400))
charts[0] & charts[1] & charts[2] & charts[3]
```

[36]:
```
alt.VConcatChart(…)
```

Yards Gained didn't show signficant separation of data from the clustering algorithms, but assessing the clusters based on isDropback has given some extremely useful insights! For all the heirarchical models, they were able to find one label that is nearly always true for isDropback. Which means based on those features, there is a way to be nearly certain that a play is a passing play. There is also a significant number of plays in this label which makes the finding more signficiant. There are 281 plays for each label that has this distiguished. That is roughly 5% of plays called for first down. This isn't a complete slam dunk but still useful! Now, I am going to set up the model for the optimal parameters found in the last and assess the labels and characteristics that influenced them.

Model Selection I am going to chose the parameters that had the widest distribution of clusters for heirarchal clustering while still having useful insights. This would be the first label generated with the ward linkage and euclidian metric.

[37]:
```
hierModel = AgglomerativeClustering(n_clusters=5, metric="euclidean", linkage="ward")
hierModel.fit(scaledFeatures)
cleanedPlays["finalLabel"] = hierModel.labels_
```

Showing yards gained for the model trained on euclidean metric and ward linkage

[38]:
```
#check to see label information again compared to isDropback and yards gained
alt.Chart(cleanedPlays).mark_bar().encode(
    x = alt.X("finalLabel:N"),
    y = alt.Y("count():Q"),
    color = "isDropback",
    tooltip=["finalLabel", 'isDropback', 'count()'],
    xOffset='isDropback:N'
).properties(
```

```
    title = "label compared with isDropback (meaning true = pass, false = run)",
    width = 700,
    height = 400)
```

[38]: alt.Chart(…)

[39]:
```
alt.Chart(cleanedPlays).mark_boxplot().encode(
    x = alt.X("finalLabel:N"),
    y = alt.Y("yardsGained:Q"),
    color = "finalLabel:N"
).properties(
    title = "label compared with yards gained",
    width = 700,
    height = 400)
```

[39]: alt.Chart(…)

Analysis I am most interested in looking into the data assigned label 4. I am also interested in looking into the data of label 2 that has 447 run plays and only 171 pass plays. For this section I plan to look at the features associated with this label and try and distinguish the factors that cause this cluster to form.
These features are offenseFormation, yardsToEndzone, receiverAlignment, secondsLeft, scoreDifference, yOffset, motionSinceLineset, and shiftSinceLineset.
Of these features, only two were in the original play data set and 6 were engineered.

[40]:
```
#taking subset of plays that are associated to label 4
label4Plays = cleanedPlays[cleanedPlays['finalLabel'] == 4]
print(label4Plays.head(3))
```

```
       gameId  playId  quarter  down  yardsToGo possessionTeam defensiveTeam  \
0  2022102302    2655        3     1         10            CIN           ATL
1  2022091809    3698        4     1         10            CIN           DAL
2  2022091112    1674        2     1         10             GB           MIN

  yardlineSide  yardlineNumber gameClock  …  motionSinceLineset  \
0          CIN              21     01:54  …                True
1          CIN               8     02:13  …               False
2           GB              25     00:35  …                True

   shiftSinceLineset secondsLeft  scoreDifference  kMeansLabel  heirLabel1  \
0              False        1014              -18            4           4
1              False         133                0            4           4
2              False        1835              -17            4           4

   heirLabel2 heirLabel3 heirLabel4  finalLabel
0           4          3          3           4
1           4          3          3           4
2           4          3          3           4
```

```
[3 rows x 58 columns]
```

Making Plots for Each Feature

[41]:
```
formationChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("offenseFormation:N", title="Formation"),
    y = alt.Y("count():Q", title="Count")
).properties(
    title="Offense Formation Counts",
    width = 300,
    height = 300
)

yardsToEndzoneChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("yardsToEndzone:N", bin=alt.Bin(extent=[0,100], step=20), title=␣
 ↪"yards to endzone bins"),
    y = alt.Y("count():Q")
).properties(
    title="Yards to Endzone Counts",
    width = 300,
    height = 300
)

receiverAlignmentChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("receiverAlignment:N"),
    y = alt.Y("count():Q")
).properties(
    title="Receiver Alignment Counts",
    width = 300,
    height = 300
)

secondsLeftChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("secondsLeft:N", bin=alt.Bin(extent=[0,3600], step=300), title=␣
 ↪"Seconds Left in game"),
    y = alt.Y("count():Q")
).properties(
    title="Seconds Left in Game Counts",
    width = 300,
    height = 300
)
motionSinceLinesetChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("motionSinceLineset:N"),
    y = alt.Y("count():Q"),
    color = "motionSinceLineset",
    tooltip=['motionSinceLineset', 'count()'],
).properties(
```

```
    title = "Motion Since Lineset Counts",
    width = 300,
    height = 300)

shiftSinceLinesetChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("shiftSinceLineset:N"),
    y = alt.Y("count():Q"),
    color = "shiftSinceLineset",
    tooltip=['shiftSinceLineset', 'count()'],
).properties(
    title = "Shift Since Lineset Counts",
    width = 300,
    height = 300)

scoreDifferenceChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("scoreDifference:N", bin=alt.Bin(extent=[-42,42], step=7), title=␣
 ↪"Point Difference in Game"),
    y = alt.Y("count():Q")
).properties(
    title="Point Difference in Game Counts",
    width = 300,
    height = 300
)

yOffsetChart = alt.Chart(label4Plays).mark_bar().encode(
    x = alt.X("yOffsetAtSnap:N", bin=alt.Bin(extent=[0,2], step=.1), title= "y␣
 ↪Offset in Game"),
    y = alt.Y("count():Q")
).properties(
    title="Y Offset in Game Counts",
    width = 300,
    height = 300
)
```

[42]:
```
#group plots together that were created above
(formationChart | receiverAlignmentChart) & (shiftSinceLinesetChart |␣
 ↪motionSinceLinesetChart) & (scoreDifferenceChart | secondsLeftChart) &␣
 ↪(yardsToEndzoneChart | yOffsetChart)
```

[42]:
```
alt.VConcatChart(…)
```

[ ]:
```

```

3.2 Analysis

The model grouped all of the empty offense formation plays together, which means there is no one but the QB in the backfield and all eligible receivers are in either a 3x2 or 4x1 position or in potentially another person is blocking on the line. The formation and receiver alignment makes

sense based on the offense formation. The part that most interests me is that players are sent in motion in this subset much more than on average. Additionally, this kind of play is run more when a team is 60-80 yards out. It makes sense that this would be run just before half time to try and get a quick score, so the seconds left in game graph is not too surprising. It is interesting that teams that are up 0-7 points run this more frequently than any other position. Maybe this means they are confident in throwing the ball and want to gain a two score lead. The box plot showed this formation as having a higher average yards gained. If the majority of plays were teams down then I would expect higher yardage per plays, but that does not seem to be the case.
Overall, this gives me good insights into the Empty play formation and how teams are allowing so many yards to it, especially since it seems clear that a pass is going to happen. The next and last cluster I am going to examine is label 2. This cluster has far more runs than passes (drop backs). I am curious to see if it has similar trends to this last label.

[43]:
```
#picking out 2nd label data
label2Plays = cleanedPlays[cleanedPlays['finalLabel'] == 2]
print(label2Plays.head(3))
```

```
        gameId  playId  quarter  down  yardsToGo possessionTeam defensiveTeam  \
3   2022100207     681        1     1         10            TEN           IND
31  2022091811     120        1     1         10             LV           ARI
41  2022091106    1380        2     1         10            MIA            NE

   yardlineSide  yardlineNumber gameClock  …  motionSinceLineset  \
3           IND              42     04:32  …                True
31           LV              44     13:00  …               False
41          MIA               8     04:05  …                True

    shiftSinceLineset secondsLeft  scoreDifference  kMeansLabel  heirLabel1  \
3               False        2972                7            4           2
31              False        3480                0            4           2
41               True        2045              -10            4           2

    heirLabel2 heirLabel3 heirLabel4  finalLabel
3            0          0          0           2
31           0          0          0           2
41           0          0          0           2

[3 rows x 58 columns]
```

[44]:
```
formationChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("offenseFormation:N", title="Formation"),
    y = alt.Y("count():Q", title="Count")
).properties(
    title="Offense Formation Counts",
    width = 300,
    height = 300
)
```

```
yardsToEndzoneChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("yardsToEndzone:N", bin=alt.Bin(extent=[0,100], step=20), title=␣
 ↪"yards to endzone bins"),
    y = alt.Y("count():Q")
).properties(
    title="Yards to Endzone Counts",
    width = 300,
    height = 300
)

receiverAlignmentChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("receiverAlignment:N"),
    y = alt.Y("count():Q")
).properties(
    title="Receiver Alignment Counts",
    width = 300,
    height = 300
)

secondsLeftChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("secondsLeft:N", bin=alt.Bin(extent=[0,3600], step=300), title=␣
 ↪"Seconds Left in game"),
    y = alt.Y("count():Q")
).properties(
    title="Seconds Left in Game Counts",
    width = 300,
    height = 300
)
motionSinceLinesetChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("motionSinceLineset:N"),
    y = alt.Y("count():Q"),
    color = "motionSinceLineset",
    tooltip=['motionSinceLineset', 'count()'],
).properties(
    title = "Motion Since Lineset Counts",
    width = 300,
    height = 300)

shiftSinceLinesetChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("shiftSinceLineset:N"),
    y = alt.Y("count():Q"),
    color = "shiftSinceLineset",
    tooltip=['shiftSinceLineset', 'count()'],
).properties(
    title = "Shift Since Lineset Counts",
    width = 300,
    height = 300)
```

```
scoreDifferenceChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("scoreDifference:N", bin=alt.Bin(extent=[-42,42], step=7), title=␣
 ↪"Point Difference in Game"),
    y = alt.Y("count():Q")
).properties(
    title="Point Difference in Game Counts",
    width = 300,
    height = 300
)

yOffsetChart = alt.Chart(label2Plays).mark_bar().encode(
    x = alt.X("yOffsetAtSnap:N", bin=alt.Bin(extent=[0,2], step=.1), title= "y␣
 ↪Offset in Game"),
    y = alt.Y("count():Q")
).properties(
    title="Y Offset in Game Counts",
    width = 300,
    height = 300
)
```

[45]:
```
(formationChart | receiverAlignmentChart) & (shiftSinceLinesetChart |␣
 ↪motionSinceLinesetChart) & (scoreDifferenceChart | secondsLeftChart) &␣
 ↪(yardsToEndzoneChart | yOffsetChart)
```

[45]:
```
alt.VConcatChart(…)
```

[46]:
```
print(cleanedPlays['offenseFormation'].
 ↪value_counts(),cleanedPlays['shiftSinceLineset'].
 ↪value_counts(),cleanedPlays['motionSinceLineset'].value_counts())
```

```
offenseFormation
SHOTGUN       2651
SINGLEBACK    2140
I_FORM         585
PISTOL         316
EMPTY          291
WILDCAT         28
JUMBO           25
Name: count, dtype: int64 shiftSinceLineset
False    3397
True     2639
Name: count, dtype: int64 motionSinceLineset
True     3405
False    2631
Name: count, dtype: int64
```

3.3 Analysis

This label has a large amount of I formation offense with a 2x1 receiver alignment. This includes all of those plays in this label. Another indicator for this cluster is a shift on the play that occurs more frequently than no shifts. This is in contrast to the first down average seen above where shifts happen less frequently than no shifts. Using shifts could be helpful in understanding if a play will be a run play. The motion data is about consistent with the average so is not signficant. The point difference is also similar to the last label so it seems that will not be a good indicator for plays. The next feature shows an increase in this label earlier in the game and slowly gets smaller as the game goes on. This could signify run plays are made more frequently early in the game and less frequently later in the game. The yards to endzone is a pretty even distribution as well and makes sense since most drives start at the teams own 25 and work their way down the field. Lastly the Y offset is more skewed left than the previous label. This could be because players are generally closer to the ball.

3.4 Conclusions

Clustering this data was interesting and showed me some new patterns that would've been tricky to find or know to look for. The 2 and 4 labels in the hierarchal cluster have given me new insights to look into and develop features around. Specifically, looking at empty formation plays and why outcomes are relatively successful when there is such a high ratio of pass to run on those plays.

The second label has given me insight into the formation that indicates a higher likelihood of run plays and also has shown a correlation between this formation and shifts pre snap. Looking more into shifts pre snap could reveal more definitive run/pass plays.

Another avenue to explore is picking a specific outcome like yardsGained and train a supervised model on the feature set created to see if it can predict play outcomes on this feature set well. I also think doing more feature engineering could be beneficial for this algorithm. I spent some time picking and choosing appropriate features, but there are many possibilities with the tracking data set. Future features to engineer could be net player orientation, defense centroid or position and in general more information about the defenses set up pre snap. This could lead to better, more unique clusters to explore.

In regards to the features I used, I could try and dumb down the impacts of offense formation and recieverAlignment on the clustering model since these both seemed like they influenced the clusters more strongly than other features. But, they are also key elements to a play call and should be involved with the clustering too. Finding a different balance there could also be interesting. This could also make other features too strong like my binary features of motion and shift since lineset. I could also try and cluster just based on the nominal features in a future iteration.

Overall, I did gain valuable insights into combinations of features that signify pass vs run on any given first down play.