NFL Combine Postion Model (Classification)

Using Machine Learning Models to Predict a Player's NFL Position Based on Combine Results

Author: Michael Helm
Session Summer B 2024

Table of Contents

Introduction

Within this introduction section we will talk about the dataset that I chose. I will also go into the background of the topic chosen to get the reader more familiarized before introducing my project and its goals.

What is the NFL Combine?

The NFL, if you do not know, is the National Football League that is held here in the United States. Note this is American football. There are 32 teams divided into two conferences: the American Football Conference (AFC) and the National Football Conference (NFC). Each conference has 4 divisions: North, East, South, and West. A season is 17 games within 18 weeks (One bye week).

Now that you have an understanding of the NFL lets get into the NFL combine. The NFL combine is officially known as the NFL Scouting Combine, it is an annual event typically held in Indianapolis, Indiana. Here college football (NCAA) players from will attempt to show their athletic abilities and skills in front of NFL scouts, coaches and general managers. The event takes place over many days, the results for a player can influence his potential in the NFL draft that follows a few months later. There are various physical test, position specific drills, college performance, measurements and more that determine a players draft stock.

Why Choose this Model?

No description has been provided for this image

The goal of this project is to create a classification model that takes in the results of the physical test that are done within the combine and try to correctly classify what position a player is based on those results. I will use the 6 drills as predictors and the response will be `Position Group` (qualitative), it is made up of the `pos` variable which we will later get into once we explore the dataset.

Predictors Definitions (Codename in original dataset):

1. 40-Yard Dash `forty`
   - The 40-yard dash is a sprint covering 40 yards (approximately 36.6 meters). It's a test of straight-line speed.
   - Seconds
2. Bench Press `bench`
   - The bench press measures upper body strength and endurance. Players lift a barbell loaded with 225 pounds (102.1 kg) as many times as they can.
   - Reps
3. 3-Cone Drill `cone`

- The 3-cone drill, also known as the "L-drill," tests a player's ability to change directions at high speed. The player runs in an L-shape pattern around three cones placed in a right-angle triangle formation.
- Seconds

4. Vertical Jump `vertical`

   - The vertical jump measures how high a player can jump from a standstill. The player jumps straight up to touch the highest possible point on a vertically mounted device.
   - Vertical inches

5. Broad Jump `broad_jump`

   - The broad jump measures a player's ability to jump as far as possible from a standing position, without any run-up. This drill evaluates lower body strength and explosiveness.
   - Horizontal inches

6. Shuttle Run `shuttle`

   - The shuttle run, often called the "5-10-5 drill," tests agility and lateral quickness. The player starts in the middle of a 10-yard area, runs 5 yards to one side, then 10 yards to the other side, and finally returns 5 yards to the starting point. Assesses change in direction.
   - Seconds

Response Definitions:

Position Group (`Position Group` will be made up of the orginal dataset variable `pos`): Output will be what position group a player is.

- Offense Positions:
  - RB (Running Back)
  - WR (Wide Receiver)
  - TE (Tight End)
  - OL (Offensive line) Will be new column created
    - OG (Offensive Guard)
    - OT (Offinsive Tackle)
    - OL (Offensive Line)
    - C (Center)
- Defense Positions:
  - S (Safety)
  - CB/DB (Corner Back / Defensive Back)
    - DB (Defensive Back)
    - CB (Corner Back)
  - LineBackers (New Column)
    - ILB (Inside Linebacker)

    - OLB (Outside Linerbacker)
  - DL (Defensive Line)
    - DE (Defensive End)
    - DT (Defensive Tackle)
    - DL (D-line)

Where Does My Data Come From?

I used Kaggle when searching for my dataset. The author is under the pseudonym DD (Dubradave) and updated the dataset about a year ago, this dataset gets yearly updated. The title is NFL Player Statistic 2002-Present. This link will take you to a kaggle source that has various datasets that deal with NFL data from 2002 to present. The specific dataset we use is called `NFL_combine_data.csv`. It contains 7000 observations and has 20 variables. As mentioned before we are only using 6 as predictors which are quantitative that are listed in the previous section.

Exploratory Data Analysis (EDA)

Exploratory Data Analysis commonly known as EDA within the community, this step takes place before we are able to do any modeling. Here we will load our data, view it, look for possible issues (including missing values) and create solutions. After we visualize through plots and tables. With those visualizations we will derive meaningful insight of the dataset to share. EDA is a crucial step of machine learning as in the step we are able to get familiar with the data, its structure, and account for meaningful relationships that might appear within the modeling.

Loading and Exploring the Dataset

We will be loading in the dataset by importing the `pandas` package that includes the function `.read_csv()` which will read in the downloaded csv file from Kaggle. It will contain combine results from 2002 till present day.

In [9]:
```
# imports package
import pandas as pd

# to read in csv
combine_results = pd.read_csv("Data for Project Combine.csv")
```

Now let's load in the first few records form the dataset using `.head()` function

In [11]:
```
#  naturally displays the first 5 observations
combine_results.head()
```

Out[11]:

	season	draft_year	draft_team	draft_round	draft_ovr	pfr_id	cfb_id	player_name
0	2002	NaN	NaN	NaN	NaN	NaN	mike-abrams-1	Mike Abrams
1	2002	NaN	NaN	NaN	NaN	NaN	jonathan-adams-1	Jonathan Adams
2	2002	NaN	NaN	NaN	NaN	AlexP.20	NaN	P.J Alexander
3	2002	NaN	NaN	NaN	NaN	NaN	jeremy-allen-1	Jeremy Allen
4	2002	2002.0	Indianapolis Colts	6.0	204.0	AlleBr20	brian-allen-2	Brian Allen

We are now able to see all the information that is offered within this dataset of NFL_combine_results loaded in as `Data for Project Combine.csv` earlier. As mentioned before it offers a total of 20 variables, out of those we will use the 6 measures of performance at the combine. Let's filter it just to include the 7 variables needed (6 predictors, 1 response) by indexing by column name . As well as create cleaner column names with `.rename()` .

- `pos` → `Position Group`
- `forty` → `Forty`
- `bench` → `Bench`
- `vertical` → `Vertical Jump`
- `broad_jump` → `Broad Jump`
- `cone` → `3-Cone Drill`
- `shuttle` → `Shuttle Run`

In [13]:
```
# reducing the dataset to only include predictors and outcome
combine_results_reduced = combine_results[['pos',
                                           'forty',
                                           'bench',
                                           'vertical',
                                           'broad_jump',
                                           'cone',
                                           'shuttle']].copy()

# renaming predictors
combine_results_reduced = combine_results_reduced.rename(columns={'pos': 'Position
                                                                  'forty':'Forty',
                                                                  'bench': 'Bench',
                                                                  'vertical': 'Vert
                                                                  'cone':'3-Cone Dr
                                                                  'shuttle': 'Shutt
```

```
                                                          'broad_jump': 'Br
combine_results_reduced.head()
```

Out[13]:

	Position Group	Forty	Bench	Vertical Jump	Broad Jump	3-Cone Drill	Shuttle Run
0	P	NaN	NaN	NaN	NaN	NaN	NaN
1	RB	4.58	15.0	36.5	120.0	6.97	4.31
2	OG	5.38	NaN	29.5	102.0	7.90	4.73
3	FB	4.59	17.0	38.0	126.0	7.06	4.49
4	RB	4.46	19.0	33.0	112.0	6.90	4.08

We now have the variables needed to continue. Notice the change in variable names to a more user friendly name. With the help of `.shape()` we are able to look at the dimensions that we are working with.

In [15]:
```
print(f'Here is the amount of (rows, columns) in our dataset currently {combine_res
```

```
Here is the amount of (rows, columns) in our dataset currently (7038, 20).
```

Missing Data and Removing Unneeded Values

Unfortunately, this dataset is not ideal for what is needed; 2 issues will be addressed. First off, there are missing values within some of the records that we will handle by only using records that have no null values. Secondly, the removal of positions not specified including `P`, `LS`, `EDGE` and `FB` in the `pos` column to ensure are model is not trained on positions that we are not looking for. We will also remove `QB` as a lot do not participate in all the drills thus creating many null values.

In [141…
```
# drops null values
combine_results_reduced.dropna(inplace=True)

# getting rid of unneeded positions
combine_results_reduced = combine_results_reduced[combine_results_reduced['Position
combine_results_reduced = combine_results_reduced[combine_results_reduced['Position
combine_results_reduced = combine_results_reduced[combine_results_reduced['Position
combine_results_reduced = combine_results_reduced[combine_results_reduced['Position

# shows dimensions
combine_results_reduced.shape #shows dimensions
```

Out[141…
```
(3072, 7)
```

Notice that we have gone from 7038 observations to 3072 observations by removing any record with missing/null values as well as moving certain position groups we are uninterested in. We signifcantly downsized the dataset ,but still have more than a sufficent amount to create a model.

We will now create 4 new variables in the column `Position Group`; `O-line` containing values `OL`, `OG`, `C`, `O`, as well as column `DB/CB` containing values `DB` and `CB`, then column `LB` containing `ILB` and `OLB` and lastly `D-line` which contains values `DE`, `DL`, `DT`.

In [143…

```
# consolidates certain positions (pos) into Position Group
combine_results_reduced['Position Group'] = combine_results_reduced['Position Group
```

The outcome variable `Position Group` will now has the possible values of:

- `O-Line`
- `D-Line`
- `LB`
- `DB/CB`
- `WR`
- `RB`
- `S`
- `TE`

Visual EDA

Within this section we will explore some visual EDA. Included is a bar graph where we take a look at amount of players per `Position Group`. Box plots that will show insight to `40-Yard Dash` time across `Position Group`. Then a correlation matrix to see which variables are positively or negatively correlated. Lastly, a metric pair plot which display a grid of 9 plots that focus on the metrics `Bench`, `Forty`, and `Vertical Jump`.

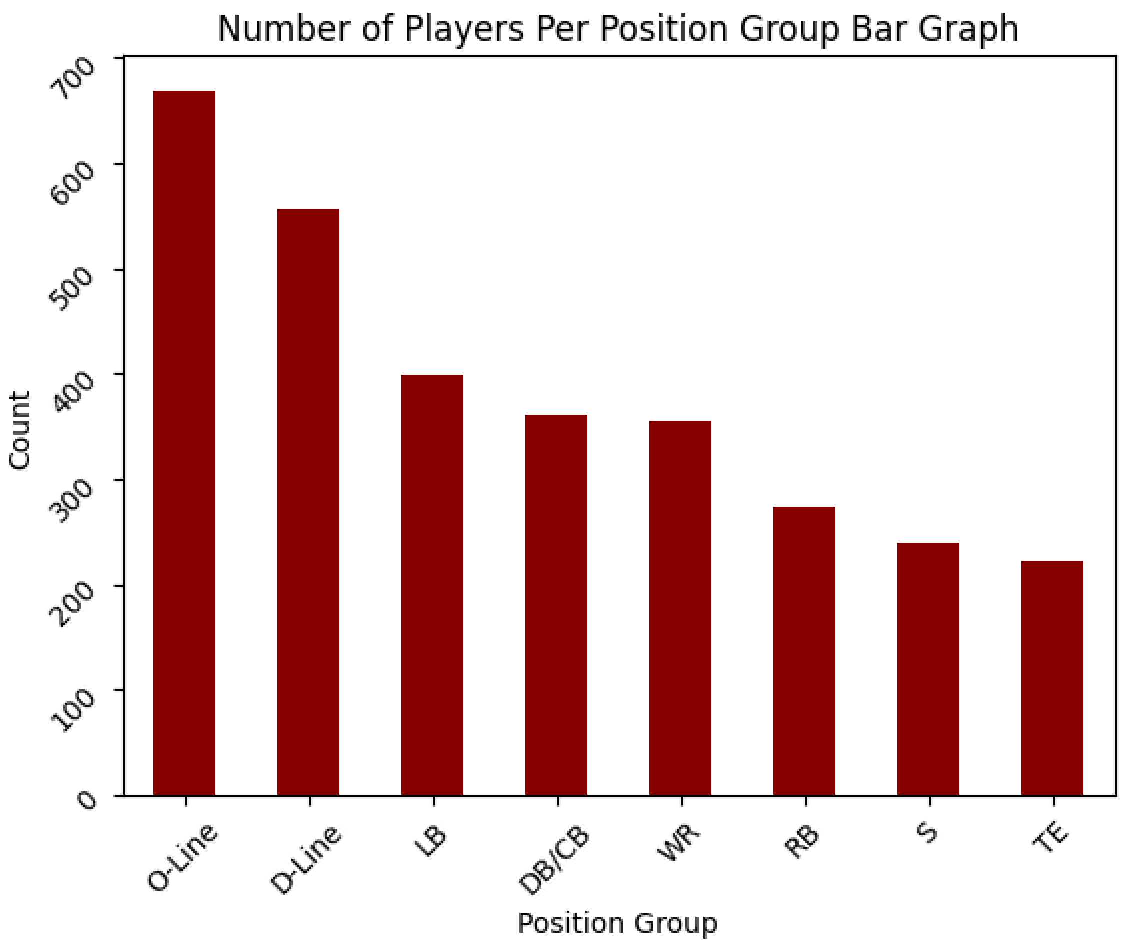
Position Bar Plot

Below is a bar graph of the amount of players per postion. As you will see there is a difference in the amount of position type across the board but there is still at least a few hundred observations per group. Used was a package called `matplotlib.plyplot` and the functions it offers allowed me to visualize a barplot.

In [32]:

```
import matplotlib.pyplot as plt
# creates a count for each position
pos_count =  combine_results_reduced['Position Group'].value_counts()
```

```
# plots the count into barplot
pos_count.plot(kind='bar', color = 'darkred')

# rest is labeling and updating the ticks to be angles
plt.xlabel('Position Group')
plt.ylabel('Count')
plt.title('Number of Players Per Position Group Bar Graph')
plt.xticks(rotation=45)
plt.yticks(rotation=45)
plt.show()
```



Key takaways:

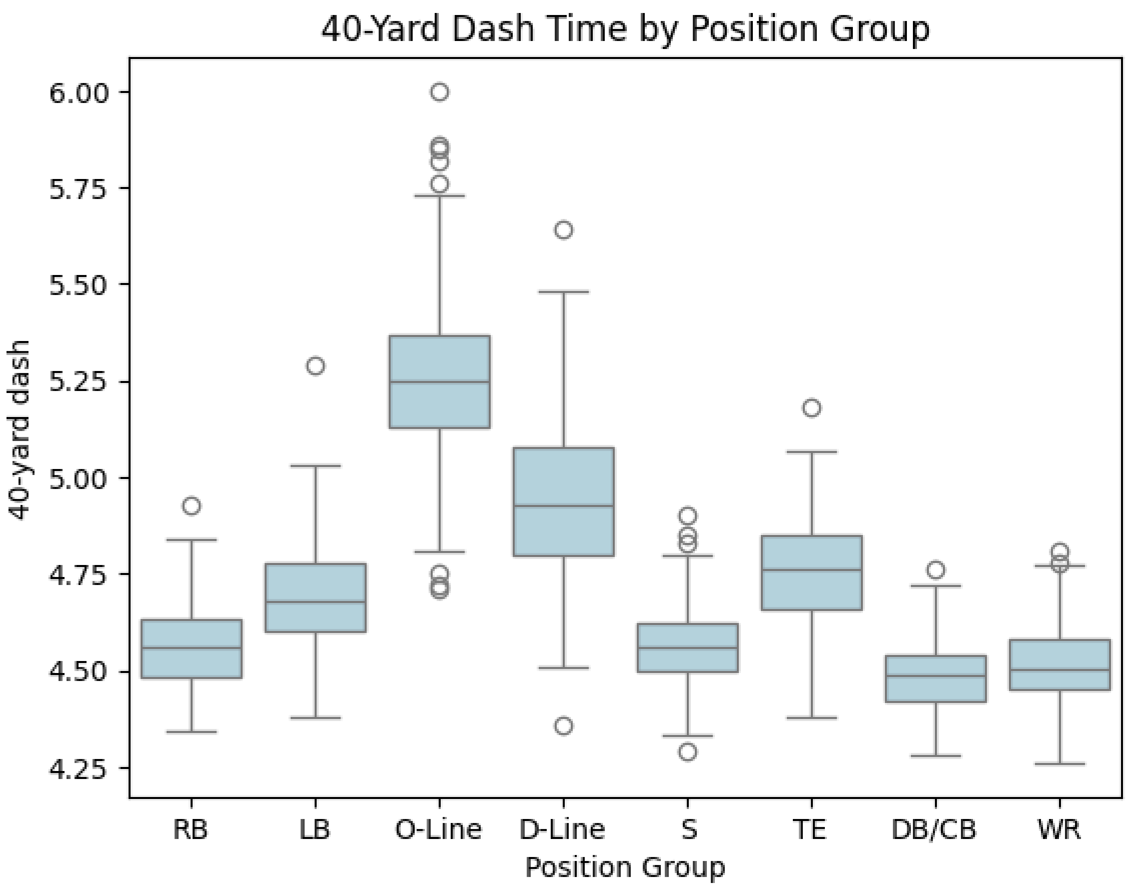
- Both of the `O-Line` (about 650) and `D-Line` (about 550) have heavier amount of player which makes sense as on a football field they combine to 10 out of 22 players on the field at once.
- The group with the least is `TE`, that is slightly above 200.
- While the rest of the positions sit somewhere between the highly listed value and the lowest value.

40 Yard Box Plot by Position

Many who watch the NFL combine love to see this event. It is an iconic and eletric event out of all the drills the players do. I thought to include a boxplot for each `Position Group` to see how their `40-Yard Dash` time compare to one another. Using a different package known as `seaborn` we are able to use the `.boxplot()` function for visualization. Let's take a look below.

In [36]:
```python
# new package seaborn
import seaborn as sb

# creates boxplot using Position Group and Forty
sb.boxplot(x = 'Position Group',
           y= 'Forty',
           data = combine_results_reduced,
           color = 'lightblue')

# labeling
plt.title('40-Yard Dash Time by Position Group')
plt.xlabel('Position Group')
plt.ylabel('40-yard dash')
```

Out[36]:
```
Text(0, 0.5, '40-yard dash')
```



Some Insight

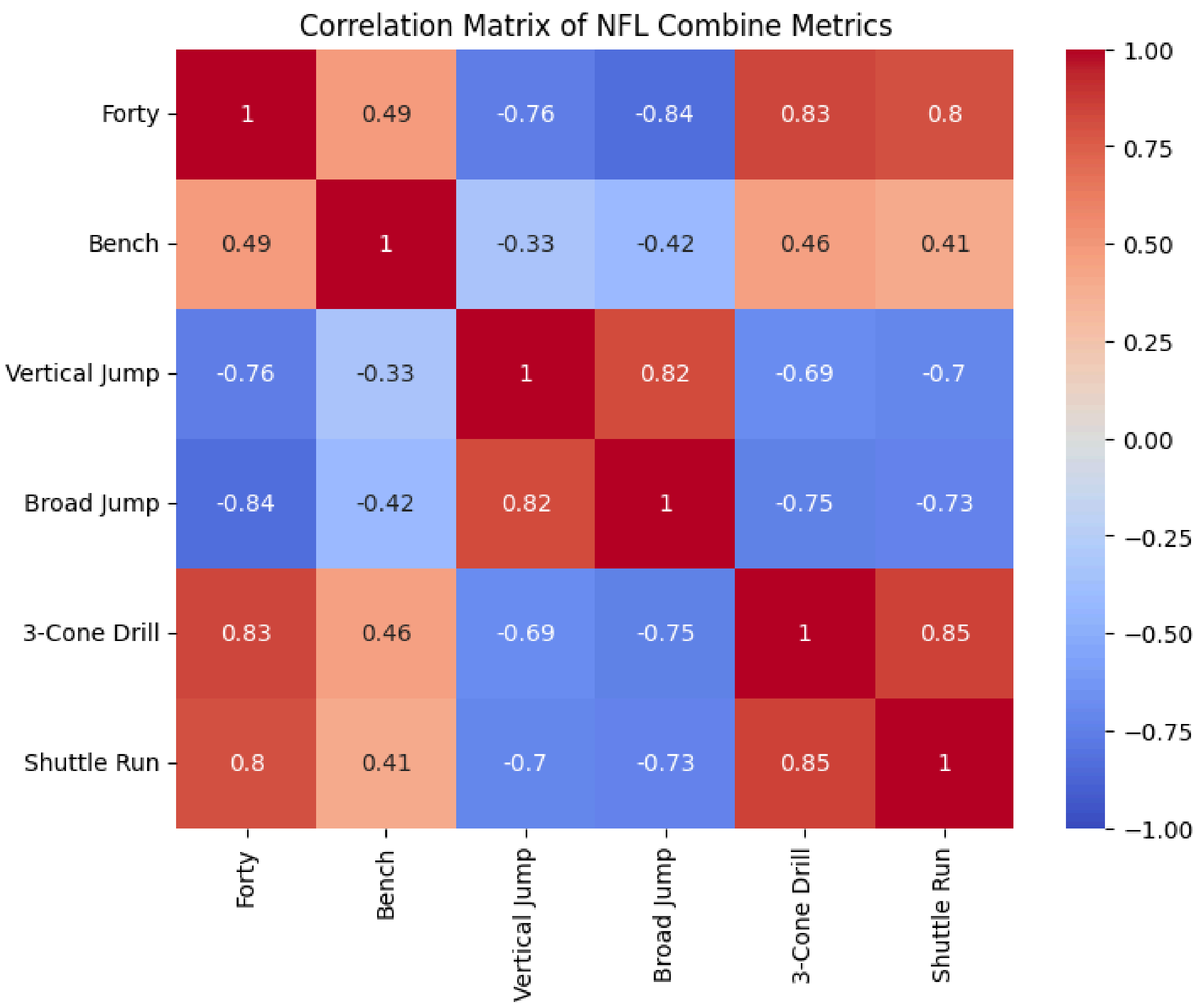
- If you look below you'll see that linemen (`O-line` and `D-line`) have slower times than the rest of the group, but with `O-line` slightly faster than `D-line` .

- `O-line` have the most variance between their times as well as extreme outliers.
- Other signifcant findings are `DB/CB` and `WR` are the top for forty times with an average around 4.50. DB/CB are slightly faster overall.

Correlation Matrix of NFL Combine Metrics

Below you will see a correlation matrix that doubles as a heatmap, values will range from -1 (Negatively Correlated) and 1 (Positively correlated). We use the heatmap as it is a visual that helps us understand relationships between variables within our dataset. We use the same package as before, `seaborn`, but this time using the `.heatmap()` function to help us visualize the matrix.

In [40]:
```
# takes in numerical values
df_numerical = combine_results_reduced.select_dtypes(include=['number'])

# puts the predictors and values into a correlation matrix
corr_matrix = df_numerical.corr()

# adjust visual size on page
plt.figure(figsize=(8, 6))

# creates a heatmap based on corr_matrix made earlier
sb.heatmap(corr_matrix,
           annot=True,
           cmap='coolwarm',
           vmin=-1,
           vmax=1)

# labeling
plt.title('Correlation Matrix of NFL Combine Metrics')
plt.show()
```



	Forty	Bench	Vertical Jump	Broad Jump	3-Cone Drill	Shuttle Run
Forty	1	0.49	-0.76	-0.84	0.83	0.8
Bench	0.49	1	-0.33	-0.42	0.46	0.41
Vertical Jump	-0.76	-0.33	1	0.82	-0.69	-0.7
Broad Jump	-0.84	-0.42	0.82	1	-0.75	-0.73
3-Cone Drill	0.83	0.46	-0.69	-0.75	1	0.85
Shuttle Run	0.8	0.41	-0.7	-0.73	0.85	1

Below is the correlations categorized by their correlation strength.

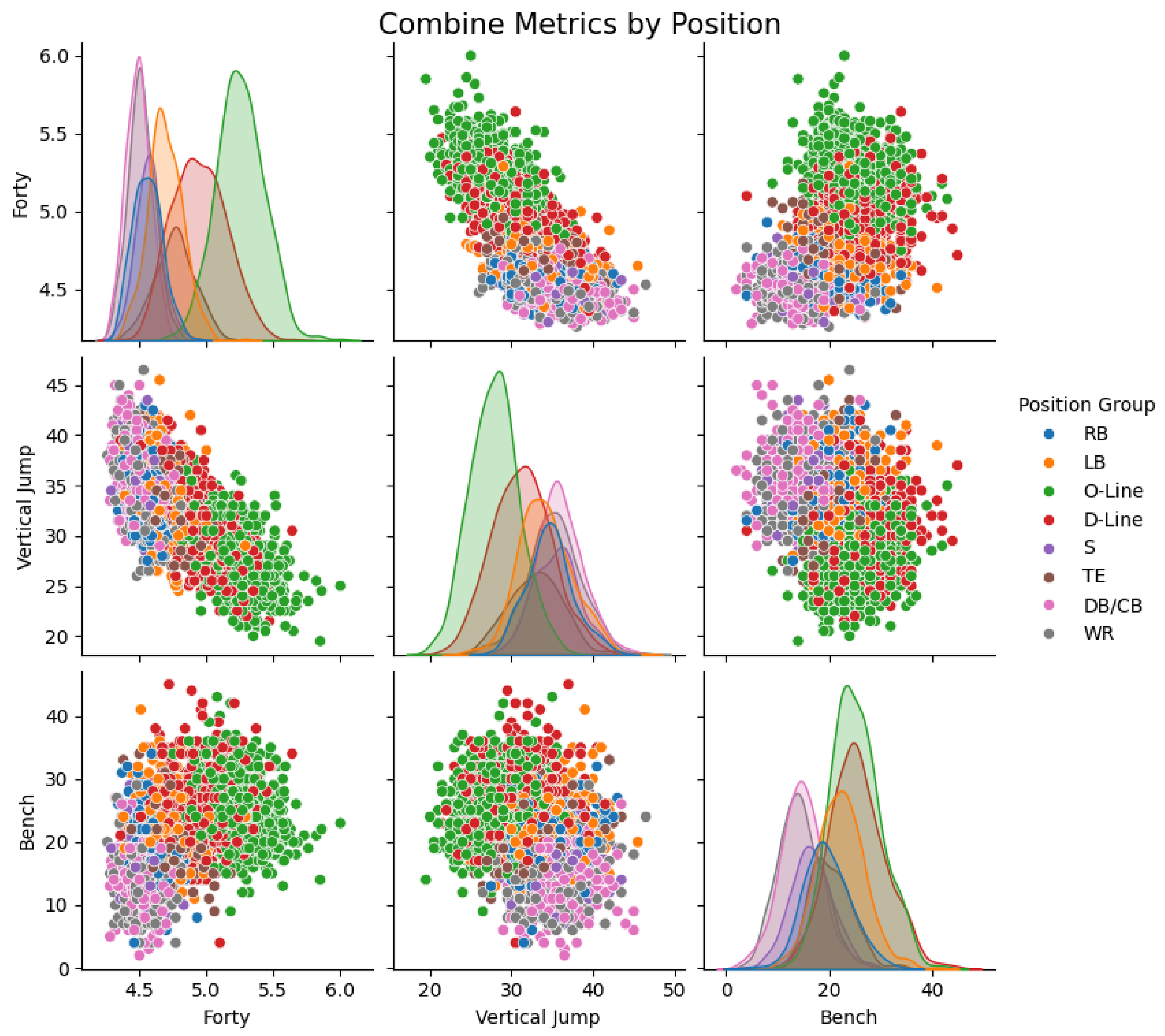
- **Positive Correlations (Show as Red above)**
  - **Moderate Correlations**
    - `Shuttle Run` : `Bench` (0.41)
    - `3-Cone Drill` : `Bench` (0.46)
    - `Bench` : `Forty` (0.49)
  - **Strong Correlations**
    - `Shuttle Run` : `Forty` (0.80)
    - `Broad Jump` : `Vertical Jump` (0.82)
    - `3-Cone Drill` : `Forty` (0.83)
    - `Shuttle Run` : `3-Cone Drill` (0.85)
- **Negative Correlations (Shown as Blue above)**
  - **Weak Correlations**
    - `Vertical Jump` : `Bench` (-0.33)
  - **Moderate Correealtions**
    - `Broad Jump` : `Bench` (-0.42)
    - `3-Cone Drill` : `Vertical Jump` (-0.69)
  - **Strong Correlations**

- `Shuttle Run` : `Vertical Jump` (-0.70)
- `Shuttle Run` : `Broad Jump` (-0.73)
- `3-Cone Drill` : `Vertical Jump` (-0.75)
- `Vertical Jump` : `Forty` (-0.76)
- `Broad Jump` : `Forty` (-0.84)

Combine Metrics Pair

Below you will see a visual that contains 6 different scatter plots that are based on 3 variables (`Forty` , `Vertical Jump` and `Bench`) as well as displaying the point (`Position Group`) by color. The 3 variables listed are the most defining events at the combine. The diagonal that you see is the distributions of the metrics seperated by position group to the right of the scatter plots you will see a legend for each position group and a corresponding color. With these scatter plots you are able to see the relationships between 2 out of the 3 variables. If you take a look further, each position group seems to have a trend within each scatter plot. Once again this is done with the use of the `seaborn` package, but with the function `.pairplot()` .

In [45]:
```
# uses seaborn function pairplot
sb.pairplot(data = combine_results_reduced, hue='Position Group', vars=['Forty', 'V

# labeling
plt.suptitle('Combine Metrics by Position', y = 1.01, fontsize=15)
plt.show()
```



Let's take a look at 2 of the plots to derive some quick insight and display how we should intepret these.

- If you look at the middle top scatter plot that compares `Forty` (y-axis) to `Vertical Jump` (x-axis) you will notice the green dots (`O-line`) are grouped together with typically a higher `Forty` time and a lower `Vertical Jump`
- If we now look at the bottom middle scatter plot that compares `Bench` (y-axis) and `Vertical Jump` (x-axis) and look at the pink dots (`DB/CB`), the whole position group seems overall to have higher `Vertical Jump` paired with low amount of 'Bench' reps.
- Something that isn't specific to one graph is that certain position groups have more variability. For example, if you look at the red points (`D-line`) on any plot you will see they have more variability then the green (`O-line`) or pink (`DB/CB`).

Model Creation

Training and Testing Split of Data

To start off in this section, we will split the data into 80% for training and 20 % for testing. This ensures we have a good amount of data to train our models on as well as test them on later. The 20% test data will be untouched until the final model to fairly assess the performance. We will set a value `random_state = 10` within `train_test_split()`, this ensures that the results of the split will be identical everytime we rerun the code. We will also use stratified sampling based on the outcome variable `Position Group` or `y` below.

In [50]:
```
# import neccessary package
from sklearn.model_selection import train_test_split

# Predictors
x = combine_results_reduced.drop(columns = ['Position Group'])

# Outcome Variable
y = combine_results_reduced['Position Group']

# creates four datasets split into test/train and then x/y
combine_x_train, combine_x_test, combine_y_train, combine_y_test = train_test_split
```

`train_test_split()` has four datasets as outcomes.

- combine_x_train will be a dataset with the predictors for training set
- combine_x_test will be a dataset with the outcomes for training set.
- combine_x_train will be a dataset with the predictors for testing set.
- combine_x_train will be a dataset with the outcomes for testing set.

Preproccessing

We will now create a recipe but in terms of python. We will use this same recipe for any model we choose to fit. As mentioned before we are only 6 predictors all which are quantitative: `Forty`, `Vertical Jump`, `Broad Jump`, `Bench`, `Shuttle Run`, `3-Cone Drill`.

In [54]:
```
# packages needed for this code block
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import LabelEncoder

# allows us to encode or categorical values which are text into numerical
label_encoder = LabelEncoder()
combine_y_train_encoded = label_encoder.fit_transform(combine_y_train)

# all predictors are numerical
numerical_predictors = ['Forty', 'Vertical Jump', 'Broad Jump',
                        'Bench', 'Shuttle Run', '3-Cone Drill']

# similar to recipe creation in r
combine_recipe = ColumnTransformer(transformers = [('num', StandardScaler(), numeri
```

Something I believe I should point out is that with using python there is not neccesarily a `recipe()` function like in Rstudio. There are still preprocessing steps similar to those in a recipe, but the outcome variable is not included with the recipe;instead it is incorporated when fitting the models. There the model determines the relationship between predictors and outcome. `StandardScaler()` is what standardize the numerical predictors similar to that of `step_scale()` and `step_center()` in R.

Model Pipelines

Within this section we will be focusing on creating different machine learning models including; Logistic Regression (Elastic Net), K-Nearest Neighbors (KNN), Random Forest and Simple Vector Machine. These models will be used to classify NFL players position based on their results from the 6 predictors at the combine. Pipelines will be created for each model previously listed.

In [58]:
```python
# import packages to create pipelines for specific models
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

# logistic pipeline
logistic_en_pipeline = Pipeline(steps=[('preprocessor', combine_recipe),
                                    ('classifier', LogisticRegression(penalty= 'ela
                                                                      solver = 'sag
                                                                      max_iter=1000
                                                                      random_state=
                                                                     l1_ratio = .5)

# K Neighbors pipeline
knn_pipeline = Pipeline(steps = [('preprocessor', combine_recipe),
                                 ('classifier', KNeighborsClassifier())])

# random forest pipeline
rf_pipeline = Pipeline(steps = [('preprocessor', combine_recipe),
                                ('classifier', RandomForestClassifier(random_state=

# support vector machine pipeline
svm_pipeline = Pipeline(steps = [('preprocessor', combine_recipe),
                                 ('classifier', SVC(probability = True, random_stat
```

The function used above is `Pipeline()` which like `workflow()` in R, from this we were able to fit the models. We also applied the `combine_recipe` to each of the pipelines for the four models, we can use these pipelines later for training, evaluation and prediction.

Note: We use Logistic Regression, but with a penalty of elastic net, thus we can change the penatly to lean toward lasso or ridge based on penalty value.

Performance Metrics of Base Models

I believe there is valuable information when fitting and viewing the performance metrics of each of the four untuned models. The two performance metrics I will be looking at will be Area Under the Curve (AUC) and Error Rate (1- accuracy). In order to pick the best model we will have to look at these simple models as well as the tuned models later on to see which we will decided to model the testing data on.

In [62]:
```
from sklearn.metrics import roc_auc_score, accuracy_score

# defining a function
def performance_metrics(pipeline_or_tunedGrid, combine_x_data, combine_y_data_encod
    fitted_model = pipeline_or_tunedGrid.fit(combine_x_data, combine_y_data_encoded

    y_pred = pipeline_or_tunedGrid.predict(combine_x_data)
    y_prob = pipeline_or_tunedGrid.predict_proba(combine_x_data)


    auc = roc_auc_score(combine_y_data_encoded, y_prob, multi_class = 'ovr')
    acc = accuracy_score(combine_y_data_encoded, y_pred)

    error_rate = 1 - acc
    print(f"{model_name} AUC: {round(auc, 8)}")
    print(f"{model_name} Error Rate: {round(error_rate, 8)}\n")
    return [auc, error_rate, fitted_model, y_pred]

log_untuned_metrics = performance_metrics(logistic_en_pipeline, combine_x_train, co
knn_untuned_metrics = performance_metrics(knn_pipeline, combine_x_train, combine_y_
rf_untuned_metrics = performance_metrics(rf_pipeline, combine_x_train, combine_y_tr
svm_untuned_metrics = performance_metrics(svm_pipeline, combine_x_train, combine_y_
```

```
Logistic Regression (Elastic Net) AUC: 0.89359043
Logistic Regression (Elastic Net) Error Rate: 0.45421245

K-Nearest Neighbors AUC: 0.93761152
K-Nearest Neighbors Error Rate: 0.37932438

Random Forest AUC: 0.99999985
Random Forest Error Rate: 0.000407

Support Vector Machine AUC: 0.90116995
Support Vector Machine Error Rate: 0.44525845
```

Instead of repeatedly writing code for each model as well as untuned and tuned I chose to define this function `performance_metrics`. It will help allow us view the two important performance metrics mentioned earlier. All we have to do is call the function once per model we want to see the performance. It prints out the metrics as well as stores them in a list for later use, `[auc, error_rate]`

Something that you see in the output is that it seems that Random Forest is the best model with an extremely low error rate of 0.000407 and almost a 1 AUC, this is misleading as there is a high chance that there is overfitting occurring with this model. We will keep in mind when we tune the model. On the other hand, the worst model seems to be Logistic Regression (Elastic Net) with the highest error rate out of the models, 0.45421245. If I had to choose a model as of now, without tuning I would choose Random Forest, but if it is indeed overfitting (like I expect) I would choose K-Nearest Neighbors as it is the second best (and practical) error rate.

GridSearchCV (Tuning) with K Fold Cross Validation

Within in this section we will use a function called `GridSearchCV()` which is similar to `tune_grid()` in R. This function gives us the ability to tune the hyperparameters of the different models. It comes from the package `scikit-learn` which we have used before. With the function you are also able to perform 5-fold cross validation as a parameter of the function. That means the data will be split into 5 sets where one is the validation and the rest are used to train the model.

Before using the function we need to set up a parameter dictionary that we will use for each grid. In the dictionary we will specify the metrics we want to test including accuracy and auc. Now, we can go ahead and apply `GridSearchCV()` to our 4 models choosen and use the parameter dictionary as an argument for `GridSearchCV()`.

In [67]:
```
from sklearn.model_selection import GridSearchCV
import numpy as np
scoring = scoring = {'accuracy': 'accuracy', 'auc': 'roc_auc_ovr'}
```

Logistic Regression (Elastic Net)

In [69]:
```
# logistic regression but elastic net params
elastic_net_params = {
    'classifier__C': [0.1, 1.0, 10.0, 100], # regularization
    'classifier__l1_ratio': [0.1, 0.5, 0.7, 1.0]  # elastic net ratio
}

# logistic regression but elastic net grid
logreg_en_search = GridSearchCV(
    estimator = logistic_en_pipeline,  # logistic regression pipeline
    param_grid = elastic_net_params,   # based on values above
    scoring = scoring,                 # scoring dictionary created earlier
    refit = 'accuracy',                     # will refit based on best auc once we
    cv = 5,                            # refers to cross validation of value 5
    return_train_score = True
)
```

Above we were able to have two parameters for the Logistic Regression:

- `classifier__C` is the amount of regularization applied to the model, there are 4 values; 0.1, 1, 10, 500 and 1000. This is done so we are able to cover a wide range of regulrization in order to determin to see which creates a better model.

- While `classifier__l1_ratio` is the elastic net ratio where we will test 4 values; 0.1, 0.5, 0.7 and 1.0. A value closer to 1 indicates a more lasso model.

K-Nearest Neighbors

In [72]:
```
knn_params = {
    'classifier__n_neighbors': [3, 5, 7, 9, 11]  # number of neighbors
}

knn_search = GridSearchCV(
    estimator = knn_pipeline, # K-nearest neighbors pipeline
    param_grid = knn_params,  # Knn params listed above
    scoring = scoring,        # scoring dictionary created earlier
    refit = 'accuracy',            # refit based best auc
    cv = 5,                   # refers to cross validation of value 5
    return_train_score = True
)
```

Above we create only a single parameter which is the amount of neighbors for K-Nearest Neighbors.

- `classifier__n_neighbors` is 5 possible values for the number of neighbors including 3, 5, 7, 9, and 11. It will test each value and return the best amount of neighbors to the model.

Random Forest

In [75]:
```
# random forest params
rf_params = {
    'classifier__n_estimators': [200, 300, 400],  # number of trees
    'classifier__max_features': [2, 3, 4, 5],  # predictors (mtry in R)
    'classifier__min_samples_split': [10, 12, 14, 16]  # samples required to split
}

# random forest grid
rf_search = GridSearchCV(
    estimator = rf_pipeline, # random forest pipeline
    param_grid = rf_params,  # params specified above
    scoring = scoring,       # scoring dictionary created earlier
    refit = 'accuracy',           # will refit based on best auc once we fit model
    cv = 5,                  # refers to cross validation of value 5
    return_train_score = True)
```

Above we have 3 parameters for Random Forest which we are familiar with including:

- `classifier__n_estimators` is the number of trees, has a total of 3 values that are 200, 300 and 400
- `classifier__max_features` is the number of predictors used in the model, it can range from 2 to 5, ideally somewhere in the middle will be the best.
- `classifier__min_samples_split` , lastly is the minimum amount of observations needed ranging from 10 to 20 by increments of 2.

Support Vector Machine

In [78]:
```
# simple vector machine params
svm_params = {
    'classifier__C': [0.1, 1, 10, 100],  # Regularization parameter
    'classifier__kernel': ['linear', 'rbf'],  # Kernel type
    'classifier__gamma': ['scale', 'auto'],  # Kernel coefficient for 'rbf'
}

# simple vector machine grid
svm_search = GridSearchCV(
    estimator = svm_pipeline, # sinmple vector machine pipeline
    param_grid = svm_params,  # based on params above
    scoring = scoring,        # scoring dictionary created earlier
    refit = 'accuracy',            # will refit based on best auc once we fit model
    cv = 5,                   # refers to cross validation of value 5
    return_train_score = True
)
```

Above we have 3 different parameters that we are using for Simple Vector Machines.

- First is the `classifier__c` parameter, which is the amount of regularization, containing of 4 values: 0.1, 1, 10 , 100.
- Second is `classifier__kernal` which chooses between a `linear` or a radial basis function (`rbf`) for more complex relationships.
- Last is the `classifier__gamma` which is need if the kernal is `rbf` , if that is the case gamma can be 1 of two values, scale or auto.

I will quickly go over some of the arguments within `GridSearchCV()` for more clarification.

- `cv` refers to cross validation. If this valued is specified like in our case with the number 5 then it will perform 5-fold cross validation. We did this for each model.
- `refit = 'accuracy'` means that once we fit the gridsearch it will take the model that creates the highest accuracy thus lowest error rate from specific hyperparameters.

Tuned Model Fitting and Performance Metrics

Let's go over what has been recently done. First, we were able to establish a baseline model with using `pipeline()` . Then, we applied `GridSearchCV()` function where we applied of pipeline models, parameters for those pipelines and specified a cross fold validation of 5. We now are calling the user-defined function `performance_metrics()` . It will fit each of the tuned models based on the best hyperparameters from the grid in order to achieve the highest accuracy for each of the 4 models. It will also calculate area under the curve (AUC) and error rate for each model.

In [83]:
```
log_tuned_metrics = performance_metrics(logreg_en_search, combine_x_train, combine_
knn_tuned_metrics = performance_metrics(knn_search, combine_x_train, combine_y_trai
rf_tuned_metrics = performance_metrics(rf_search, combine_x_train, combine_y_train_
svm_tuned_metrics = performance_metrics(svm_search, combine_x_train, combine_y_trai
```

```
Logistic Regression AUC: 0.89359043
Logistic Regression Error Rate: 0.45421245

K-Nearest Neighbors AUC: 0.91727204
K-Nearest Neighbors Error Rate: 0.41758242

Random Forest AUC: 0.97915348
Random Forest Error Rate: 0.21733822

Support Vector Machine AUC: 0.8947542
Support Vector Machine Error Rate: 0.45665446
```

The output above is the AUC and error rate of each model. Remember these were done on the training data, no testing at this time has been done yet. For classification problems the metrics we care about are AUC and error rate. The higher the AUC the better, the lower the error rate the better.

- Once again Random Forest is the best model, but both metrics have gotten worse from the untuned model. AUC decreased and error rate increased to 0.21733. This makes more sense and makes me less skeptical that the tuned model is overfitting.
- The second model out of then tuned models I would just would be the K-Nearest Neighbors as it has the second best AUC and error rate.
- After the tunning the worst is Support Vector Machine as it has the lowest AUC and error rate of 0.4566, I will automatically exclude this model for the final testing.

In [85]:
```
print("Best parameters for Logistic Regression:", logreg_en_search.best_params_)

print("Best parameters for K-Nearest Neighbors:", knn_search.best_params_)

print("Best parameters for Random Forest:", rf_search.best_params_)

print("Best parameters for SVM:", svm_search.best_params_)
```

```
Best parameters for Logistic Regression: {'classifier__C': 1.0, 'classifier__l1_rati
o': 0.5}
Best parameters for K-Nearest Neighbors: {'classifier__n_neighbors': 11}
Best parameters for Random Forest: {'classifier__max_features': 3, 'classifier__min_
samples_split': 16, 'classifier__n_estimators': 200}
Best parameters for SVM: {'classifier__C': 1, 'classifier__gamma': 'scale', 'classif
ier__kernel': 'linear'}
```

Above we take a look at the best hyperparameters from the `GridSearchCV()` of each model. You will see the specific hyperparameters (from the set of parameters earlier) of what is considered the best model for each of the 4 models; Logistic Regression (Elastic Net), K-Nearest Neighbors, Random Forest and Simple Vector Machine.

Tuned VS. Untuned Performance Metrics Visualized

Within this section we will be visualizing the metrics in two seperate bar plots. They will be grouped bar plots; one will compare AUC values (tuned vs. untuned), the second will compare error rate values (tuned vs untuned) for each of the four models we have been working with. I had attempted to create this type of grouped barplot, but kept on getting stuck so I look towards chat gpt and fed it this prompt below.

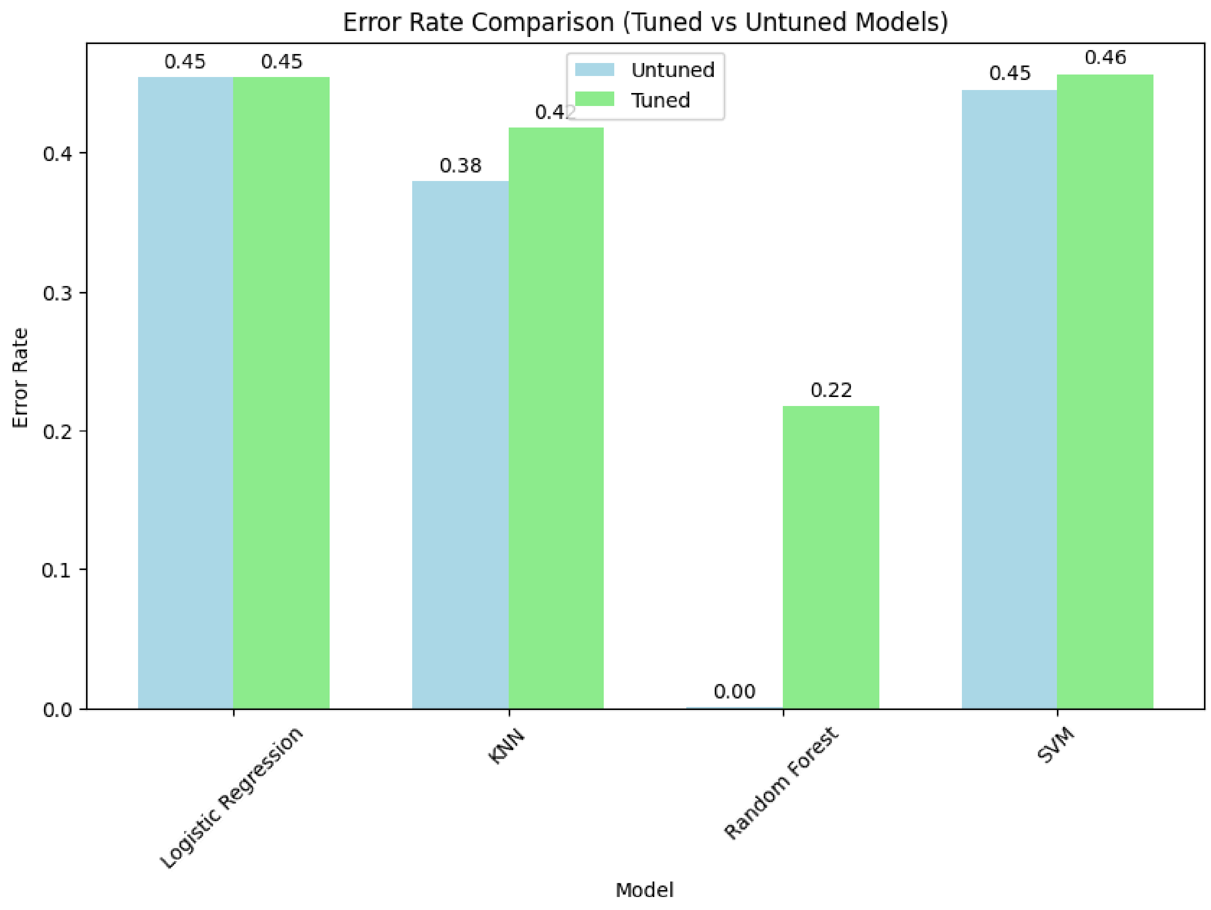
Chat GPT Prompt

I'd like two separate grouped bar plots. The first plot should compare the AUC values for four models: Logistic Regression, K-Nearest Neighbors (KNN), Random Forest, and Support Vector Machine (SVM). Each model should have two bars side by side: one for the untuned version and one for the tuned version. The second plot should do the same but compare the error rates of the tuned vs. untuned models. Please label the axes, include a legend to show which bars represent tuned and untuned models, and display the AUC and error rate values on top of the bars.

I then fed it my user-defined function to give it more context to the prompt. Below I had to update the `untuned_auc`, `untuned_Error`, `tuned_auc` and `tuned_error_rate` to my actual values as it gave me random example values at first.

Error Rate Comparison

In [95]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data for untuned and tuned error rates
models = ['Logistic Regression', 'KNN', 'Random Forest', 'SVM']

untuned_auc = [log_untuned_metrics[0], knn_untuned_metrics[0], rf_untuned_metrics[0
untuned_error_rate = [log_untuned_metrics[1], knn_untuned_metrics[1], rf_untuned_me

# Tuned metrics
tuned_auc = [log_tuned_metrics[0], knn_tuned_metrics[0], rf_tuned_metrics[0], svm_t
tuned_error_rate = [log_tuned_metrics[1], knn_tuned_metrics[1], rf_tuned_metrics[1]

# Set bar width and positions for bars
bar_width = 0.35
index = np.arange(len(models))

# Create the figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the bars for untuned and tuned error rates
bars1 = ax.bar(index, untuned_error_rate, bar_width, label='Untuned', color='lightb
bars2 = ax.bar(index + bar_width, tuned_error_rate, bar_width, label='Tuned', color

# Add labels, title, and legend
ax.set_xlabel('Model')
ax.set_ylabel('Error Rate')
ax.set_title('Error Rate Comparison (Tuned vs Untuned Models)')
ax.set_xticks(index + bar_width / 2)
ax.set_xticklabels(models, rotation=45)
ax.legend()

# Add labels on top of the bars
for bar in bars1:
    height = bar.get_height()
    ax.annotate(f'{height:.2f}', xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 3), textcoords="offset points", ha='center', va='bottom'

for bar in bars2:
    height = bar.get_height()
    ax.annotate(f'{height:.2f}', xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 3), textcoords="offset points", ha='center', va='bottom'
```



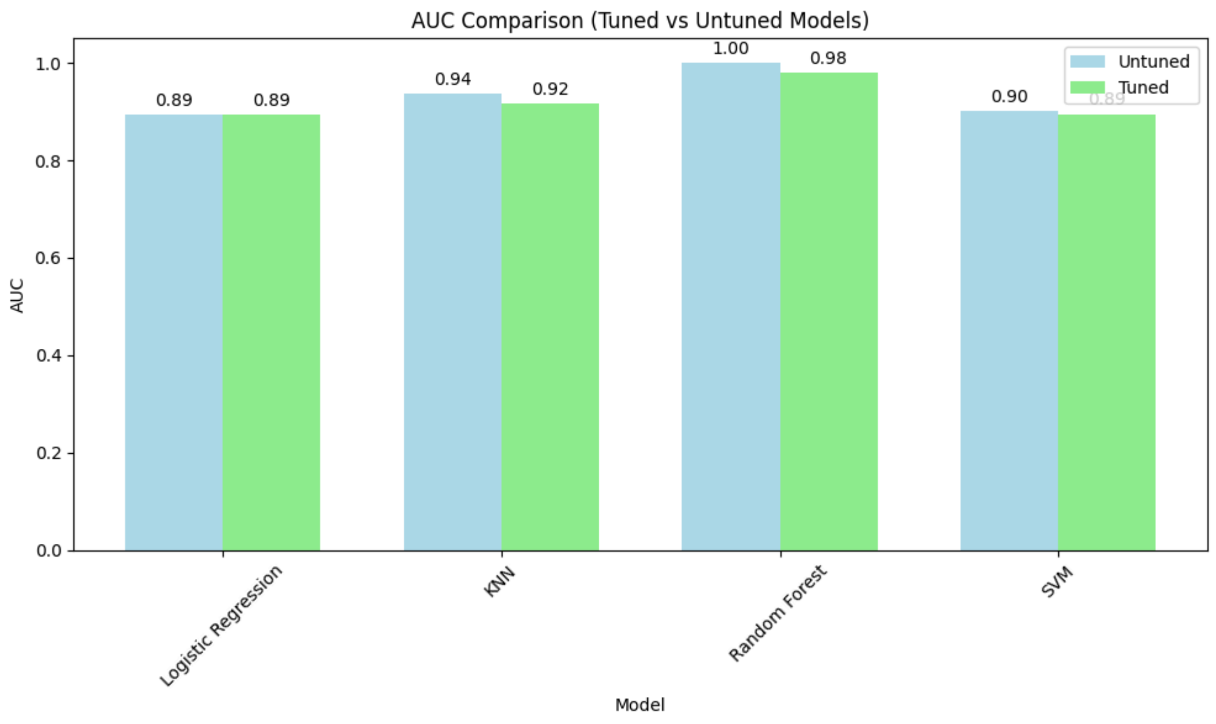
Above you will see the comparison in error rates for each of the model both tuned and untuned. From this we can make key takeaways.

- Logistic Regression (Elastic Net) stayed the same at an error rate of 0.45, there was no change from untuned to tuned model.
- K-Nearest Neighbors saw an increase in error rate by 0.4 this mean the orginal model conditions (untuned) were better than the range of parameters I gave it.
- Radom forest might be the most interesting as it was an error rate originally of 0 and jumped to 0.22. This is a good thing, but may seem counter intuitive. The reason is that originally it was overfitting and now the tuned model has a more appropiate error rate.

- Lastly the Simple Vector Machine has a slightly higher error rate by 0.01.

AUC Comparison

In [98]:
```
# Set the bar width and index positions
bar_width = 0.35
index = np.arange(len(models))

# Create the figure and axis
fig, ax = plt.subplots(figsize=(10, 6))

# Plot the bars for untuned and tuned AUC
bars1 = ax.bar(index, untuned_auc, bar_width, label='Untuned', color='lightblue')
bars2 = ax.bar(index + bar_width, tuned_auc, bar_width, label='Tuned', color='light

# Add labels and title
ax.set_xlabel('Model')
ax.set_ylabel('AUC')
ax.set_title('AUC Comparison (Tuned vs Untuned Models)')

# Set xticks and xticklabels
ax.set_xticks(index + bar_width / 2)
ax.set_xticklabels(models, rotation=45)

# Add the legend
ax.legend()

# Add labels on top of the bars
for bar in bars1:
    height = bar.get_height()
    ax.annotate(f'{height:.2f}', xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 3), textcoords="offset points", ha='center', va='bottom'

for bar in bars2:
    height = bar.get_height()
    ax.annotate(f'{height:.2f}', xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 3), textcoords="offset points", ha='center', va='bottom'

# Ensure layout is tight so nothing is cut off
plt.tight_layout()

# Show the plot
plt.show()
```



Above is the output for the grouped barplot for AUC comparison. As you can see all the models have a good AUC both before and after they are tuned. Some noticable changes between the untuned and tuned model is.

- The Logistic Regression (elastic net) has stayed the same before and after it was tuned. That means that the penalty of 0.5 was the most ideal for the model.
- K-Nearest Neighbors saw a slight decrease after tuned meaning the values we specified in the parameters dictionary weren't ideal compared to the values ingrained in the orginal model.
- Random Forest also saw a slight decrease but this is a good thing as it was overfitting before with AUC of 1. Now hopefully we got rid of that issue.
- There was practicaly no change in the Simple Machine Model.

Best Model Selection and Testing

Within this section we are nearing the end, here we will test the best model we had from the training data. Our winner is ... TUNED RANDOM FOREST MODEL. I decided this decision based on the AUC and error rate values were the best out of all 4 models. The one caveat that I should mention is that the untuned Random Forest model on technicality was the best, but I suspect overfitting and thus decided to not use it.

Here is the specific hyperparameters of our winner!

In [114…
```
rf_search.best_params_
```

```
Out[114…  {'classifier__max_features': 4,
           'classifier__min_samples_split': 16,
           'classifier__n_estimators': 200}
```

Let's get to modeling!

```
In [106…  combine_y_test_encoded = label_encoder.transform(combine_y_test)
```

```
In [112…  results = performance_metrics(rf_search, combine_x_test, combine_y_test_encoded, "T
```

```
Tuned Random Forest AUC: 0.97819722
Tuned Random Forest Error Rate: 0.2195122
```

Above we fit, predicted and gave the performance metrics.

Lets take a look put data into a table to view comparisons of actual vs. predicted values.

```
In [128…  # turning numerical values back to labels
          y_pred_test_original = label_encoder.inverse_transform(results[3])

          # real labels vs predicted in df
          comparison_df = pd.DataFrame({
              'Actual Label': combine_y_test,
              'Predicted Label': y_pred_test_original
          })

          print(comparison_df.head(20))
```

```
     Original Label Predicted Label
5993             TE              LB
2833         O-Line          O-Line
2640             RB              RB
3508          DB/CB           DB/CB
3480             WR              WR
6635          DB/CB           DB/CB
5525          DB/CB           DB/CB
2993         D-Line          D-Line
835               S              WR
6122         O-Line          O-Line
4409             LB              LB
3700             WR              WR
1258         O-Line          O-Line
3959         O-Line          O-Line
714          O-Line          O-Line
1240          DB/CB           DB/CB
518          D-Line          D-Line
1236         O-Line          O-Line
3895         O-Line          O-Line
5090         O-Line          O-Line
```

As you can see we were able to fit the tuned random forest model with the specified hyperparameters from earlier. It it almost identical to how our training set in how it performed with AUC (0.97819722) and error rate (0.2195122). Directly above we also compared the first 20 predictions (to save space) to the actual data. As you can see we only missed 2 observations, this was a high quality model.

- 5993 where it predicted TE and should have been LB. This makes sense though as these two positions normally have similar sized players.
- The other being 835 who was predicted to be a WR when it should have been S. Likewise as before a lot of the times S and WR have similar metrics in combine.

Overall I am happy with the results. Better than I orginally had thought with roughly 80% accuracy.

Conclusion

In this project I was able to demonstrate how machine learning models can be used to predict a topic that I'm fond of. I chose to do prediction of the position group of NFL players based on their performance at the NFL Combine. We began this project with a brief introduction and then into exploratory data analysis (EDA) where we went through different visuals to establish some understanding the data before creating models. We then split the data 80/20 (training/testing) and began fitting our models. We created pipelines for Logistic Regression with elastic net penalty, K-Nearest Neighbors, Random Forest and Simple Vector Machines. The tuned models performed 5-fold cross validation. We fit these models untuned and tuned (given a set of hyperparameter). The tuned models we fit based on the best parameters that gave the highest accuracy and thus lowest error rate (1 - accuracy).

Looking back to the beginning where I decided to group the player positions I believe was good idea to simplify my model and help improve it's performance. Whilst all the models showed good potential, the tuned Random Forest Model was the best in my eyes. When we tuned the Random Forest model it seemed to get rid of the suspected issue of overfitting of the untuned model. I was suprised to see how low the error rate was. When we tested the model it performed almost exactly similar to the training model based on the testing metrics of AUC and error rate. The worst model with training data was Simple Vector Machine Model which was expected. In the future if I were to add onto this project I would include additional predictors like player-specific attribbutes like height and weight. Other things that you could add would be the players college performance statistcs. I would also like to look into how the statistics of each of the metrics vary from year to year based on position.

```
In [ ]:
```