imports

In [1]:
```
import time
from pprint import pprint
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
sns.set(style="darkgrid")
sns.set_palette("bright")
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from torch import nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch
import scipy.stats as st
```

code

In [2]:
```
bookStart = time.time()
def vis_null(df):
    sns.heatmap(df.isnull(), cbar=False, cmap="YlGnBu_r")
    plt.show()

def standardize(X):
    standy = (X - np.mean(X, axis=0))/np.std(X,axis=0)
    return standy

def map_positions(df, print_dict=False):
    posList = list(df.Position.unique())
    posDict = {posList[i]:float(i) for i in range(len(posList))}
    df['Position'] = df['Position'].map(posDict).astype('float')
    if print_dict:
        print('Position Mappings: {}'.format(posDict))
    return df

def convert_to_array(df, print_info=False):
    X = df.iloc[:,1:].to_numpy(dtype='float')
    b = df.loc[:,'Position'].to_numpy(dtype='float')
    if print_info:
        print('type: \n\tX: {}\n\tb: {}'.format(type(X), type(b)))
        print('dtype: \n\tX: {}\n\tb: {}'.format(X.dtype, b.dtype))
        print('shape: \n\tX: {}\n\tb: {}'.format(X.shape, b.shape))
    return (X, b)

def create_test_train_split(X, b, test_size=0.33, random=False):
    if random:
        output = train_test_split(X, b, test_size=test_size, random_state=None
)
    else:
        output = train_test_split(X, b, test_size=test_size, random_state=27)
    return output

def apply_PCA(x, n_components=1):
    shift = x - np.mean(x, axis=0) # normalized data
    u, s, v = np.linalg.svd(shift, compute_uv=True, full_matrices=False)
    v = v.T
    return np.dot(x, v[:,:n_components])

def plot_2D_proj(X, labels=None):
    fig = plt.figure()
    proj_2d_data = apply_PCA(X, 2)
    fig.gca().scatter(proj_2d_data[:,0],proj_2d_data[:,1],c=labels)
    fig.gca().set_title('PCA 2D transformation')
    return fig

def plot_3D_proj(X, labels=None):
    fig = plt.figure()
    ax = fig.add_subplot(111,projection='3d')
    proj_3d_data = apply_PCA(X, 3)
    ax.scatter(proj_3d_data[:,0],proj_3d_data[:,1],proj_3d_data[:,2],c=labels)
    fig.gca().set_title('PCA 3D transformation')
    return fig
```

In [3]:
```
class SVM():
    def __init__(self, X, b, test_size=0.33, random=False):
        self.X = X
        self.b = b
        self.trainX, self.testX, self.trainb, self.testb = create_test_train_s
plit(self.X, self.b, test_size=test_size, random=random)

    def create_fit(self, kernel='rbf', c=1.0, dfs='ovr', gamma='auto'):
        self.clf = SVC(kernel=kernel, decision_function_shape=dfs, gamma=gamma
, C=c)
        self.clf.fit(self.trainX, self.trainb)

    def calc_performance(self, X, b):
        preds = self.clf.predict(X)
        acc = (b == preds).sum()/float(X.shape[0])
        return acc
```

In [4]:
```
def svm_helper(C, gamma, printq=True):
    svm = SVM(X,b)
    performance_dict = {}
    for kern in ['poly','rbf','sigmoid']:
        svm.create_fit(kernel=kern, c=C, gamma=gamma)
        train_acc = svm.calc_performance(svm.trainX,svm.trainb)
        test_acc = svm.calc_performance(svm.testX,svm.testb)
        performance_dict[kern+'_train'] = train_acc
        performance_dict[kern+'_test'] = test_acc
    results = performance_dict
    if printq:
        sns.barplot(y=list(results.keys()),x=list(results.values())).set_xlabe
l('Accuracy Score')
        plt.ylabel('Kernal_Split')
        pprint(results)
    return results
```

In [5]:
```
class RandomForest():
    def __init__(self, X, b, random=False):
        self.X = X
        self.b = b
        self.trainX, self.testX, self.trainb, self.testb = create_test_train_s
plit(self.X, self.b, random=random)

    def create_fit(self, bag=True, n_estimators=100, max_features='sqrt'):
        if bag:
            self.clf = RandomForestClassifier(n_estimators=n_estimators, max_f
eatures=max_features)
            self.clf.fit(self.trainX, self.trainb)
        else: # boosting
            self.clf = AdaBoostClassifier(n_estimators=n_estimators)
            self.clf.fit(self.trainX, self.trainb)

    def calc_performance(self, X, b):
        preds = self.clf.predict(X)
        acc = (b == preds).sum()/float(X.shape[0])
        return acc
```

In [6]:
```
def rf_helper(bag, max_features, PCA=False):
    rf = RandomForest(X, b)
    performance_dict = {}
    train = []
    test = []
    for i in range(1,101):
        rf.create_fit(bag=bag, n_estimators=i, max_features=max_features)
        train.append(rf.calc_performance(rf.trainX,rf.trainb))
        test.append(rf.calc_performance(rf.testX,rf.testb))
    performance_dict['train'] = train
    performance_dict['test'] = test
    results = pd.DataFrame.from_dict(performance_dict)
    sns.lineplot(data=results, dashes=False).set_xlabel('# of Estimators')
    print('Max test score: {}'.format(results['test'].max()))
    plt.ylabel('Accuracy Score')
```

In [7]:
```
class Net(torch.nn.Module):
    def __init__(self, X, b, n_hidden, n_output, activ_func='relu', random=Fal
se):
        self.activ_func = activ_func
        self.n_hidden = n_hidden
        super(Net, self).__init__()
        self.X = X
        self.b = b
        self.trainX, self.testX, self.trainb, self.testb = create_test_train_s
plit(self.X, self.b, random=random)
        self.trainX = Variable(torch.from_numpy(self.trainX).float())
        self.testX = Variable(torch.from_numpy(self.testX).float())
        self.trainb = Variable(torch.from_numpy(self.trainb).float())
        self.testb = Variable(torch.from_numpy(self.testb).float())
        self.finalTestX = Variable(torch.from_numpy(Xho).float())
        self.finalTestb = Variable(torch.from_numpy(bho).float())
        self.hidden1 = torch.nn.Linear(X.shape[1], n_hidden[0])
        self.hidden2 = torch.nn.Linear(n_hidden[0], n_hidden[1])
        if len(n_hidden) >= 3:
            self.hidden3 = torch.nn.Linear(n_hidden[1], n_hidden[2])
        if len(n_hidden) >= 4:
            self.hidden4 = torch.nn.Linear(n_hidden[2], n_hidden[3])
        if len(n_hidden) >= 5:
            self.hidden5 = torch.nn.Linear(n_hidden[3], n_hidden[4])
        self.out = torch.nn.Linear(n_hidden[len(n_hidden)-1], n_output)   # ou
tput layer

    def forward(self, x):
        if self.activ_func == 'relu':
            x = F.relu(self.hidden1(x)) # activation function for hidden layer
            x = F.relu(self.hidden2(x))
            if len(self.n_hidden) >= 3:
                x = F.relu(self.hidden3(x))
            if len(self.n_hidden) >= 4:
                x = F.relu(self.hidden4(x))
            if len(self.n_hidden) >= 5:
                x = F.relu(self.hidden5(x))
        elif self.activ_func == 'sigmoid':
            x = (self.hidden1(x).sigmoid()) # activation function for hidden l
ayer
            x = (self.hidden2(x).sigmoid())
            if len(self.n_hidden) >= 3:
                x = (self.hidden3(x).sigmoid())
            if len(self.n_hidden) >= 4:
                x = (self.hidden4(x).sigmoid())
            if len(self.n_hidden) >= 5:
                x = (self.hidden5(x).sigmoid())
        x = self.out(x)
        return x

    def train_net(self, num_epochs=1000, print_iters=False):
        performance_dict = {}
        train = []
        test = []
        x = self.trainX
        y = self.trainb
```

```
        start = time.time()
        for t in range(num_epochs):
            out = net(x)
            loss = loss_func(out, y.long())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if t >= 0:
                train_acc = self.test_net(test_unseen=False,out=out,y=y)
                train.append(train_acc)
                test.append(self.test_net(test_unseen=True))
                if t % 100 == 0 and print_iters:
                    print('iteration: {}'.format(t))
                    print('train accuracy: {}'.format(train_acc))
        performance_dict['train'] = train
        performance_dict['test'] = test
        results = pd.DataFrame.from_dict(performance_dict)
        return (results, time.time()-start, out)

    def test_net(self, out='', y='', test_unseen=False, test_final=False):
        if test_unseen:
            y = self.testb
            out = net(self.testX)
        elif test_final:
            y = self.finalTestb
            out = net(self.finalTestX)
        prediction = torch.max(out, 1)[1]
        pred_y = prediction.data.numpy()
        accuracy = float((pred_y == y.data.numpy()).astype(int).sum()) / float
(y.data.numpy().size)
        return accuracy
```

Predicting Player Position in FIFA19 using Multiclass Classification

Grant Cloud

CS 4641 - Final Project

Table of Contents

Introduction to the Dataset

The FIFA 19 dataset is sourced from Kaggle[1] and then wrangled using the submitted file fifaWrangler.py.

In [8]:
```
df = pd.read_csv('fifa19wr.csv', index_col=0)
print('df Shape:                {}'.format(df.shape))
print('Number of Features:      {}'.format(df.shape[1]-1))
print('Number of Unique Players: {}'.format(df.shape[0]))
print('Feature Set: \n',df.columns[1:])
```

```
df Shape:                (14087, 36)
Number of Features:      35
Number of Unique Players: 14087
Feature Set:
 Index(['Age', 'Overall', 'Preferred Foot', 'Body Type', 'Height', 'Weight',
       'Crossing', 'Finishing', 'HeadingAccuracy', 'ShortPassing', 'Volleys',
       'Dribbling', 'Curve', 'FKAccuracy', 'LongPassing', 'BallControl',
       'Acceleration', 'SprintSpeed', 'Agility', 'Reactions', 'Balance',
       'ShotPower', 'Jumping', 'Stamina', 'Strength', 'LongShots',
       'Aggression', 'Interceptions', 'Positioning', 'Vision', 'Penalties',
       'Composure', 'Marking', 'StandingTackle', 'SlidingTackle'],
      dtype='object')
```

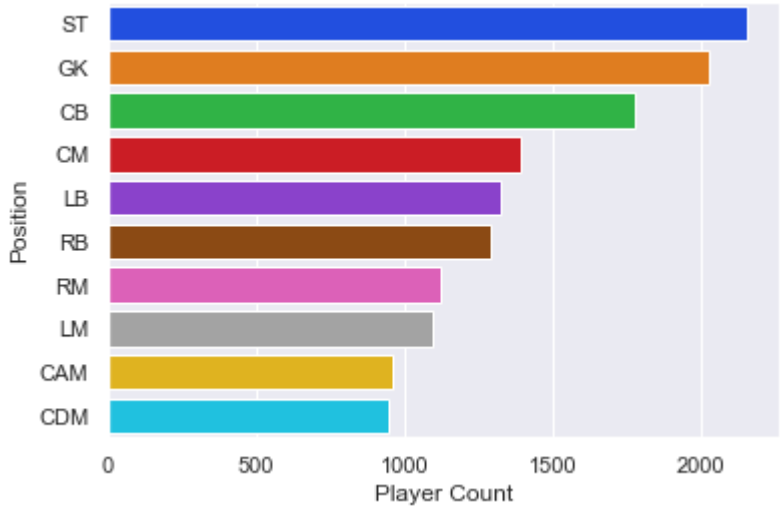
The wrangled dataset (fifa19wr.csv) contains 14087 unique players from the video game FIFA 19. Each player has 35 skills (features) with values ranging from 0.0 to 100.0 (other than Body Type and Preferred Foot which are label encoded and binary encoded respectively). Higher skill values correspond to higher performance when performing an act that involves that skill. For example, a player with 99.0 pass accuracy will outperform a player with 1.0 pass accuracy when passing.

The dataset also contains a position feature that tells us each player's position.

In [9]:
```
print('Number of Unique Positions: {}'.format(df.Position.value_counts().shape[0]))
sns.countplot(y=df['Position'], order=df.Position.value_counts().index).set_xlabel('Player Count')
```

```
Number of Unique Positions: 10
```

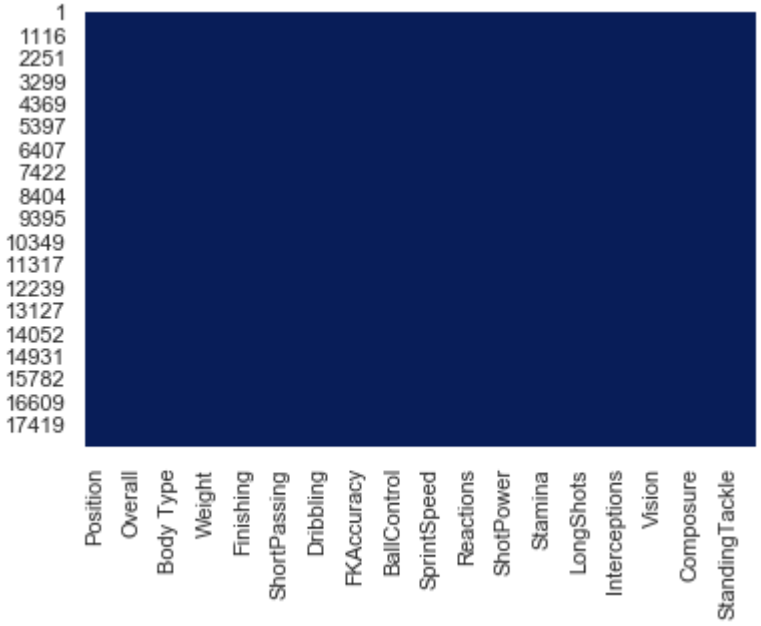
Out[9]:
```
Text(0.5, 0, 'Player Count')
```



There are 10 unique positions in this dataset and the positions correspond to the top 10 positions in the original dataset based on the number of players who play that position. Each position in this dataset contains over 900 instances, which means there is plenty of data to train and test on.

In [10]:
```
print('null values (white=null):')
vis_null(df) # show null values in the dataframe
```

```
null values (white=null):
```



Since the data has already been wrangled, there are no null or missing values in the dataset.

In order to perform machine learning on the dataset, the position column is label encoded and the dataframe is converted to a numpy array. The position encoding dictionary can be seen below.

In [11]:
```
posMap = df.Position.value_counts().index
df = map_positions(df, print_dict=True) # 'ST' --> 1.0
X, b = convert_to_array(df, print_info=True) # pd.df --> np.array
```

```
Position Mappings: {'ST': 0.0, 'GK': 1.0, 'CB': 2.0, 'CAM': 3.0, 'CDM': 4.0,
'RM': 5.0, 'LM': 6.0, 'LB': 7.0, 'CM': 8.0, 'RB': 9.0}
type:
        X: <class 'numpy.ndarray'>
        b: <class 'numpy.ndarray'>
dtype:
        X: float64
        b: float64
shape:
        X: (14087, 35)
        b: (14087,)
```

The supervised learning problem is predicting player position based on their skills, which makes this a multiclass classification problem. This is a fun and interesting problem for three main reasons:

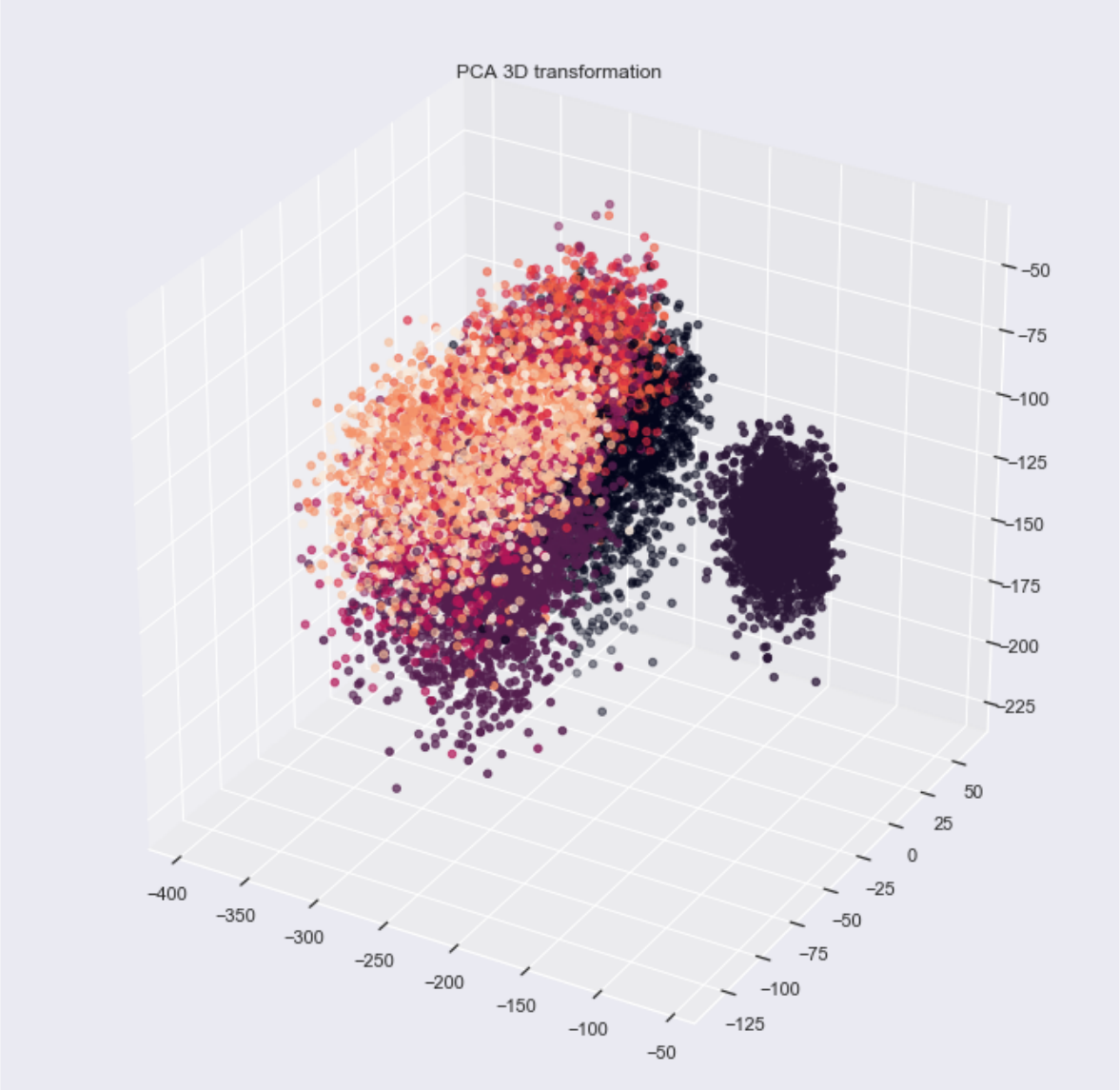
- Large dataset size so it is non-trivial (over 1000 datapoints, more than 5 features, non-trivial distribution)
- I'm an avid FIFA player so I've played with many of these players and positions and have seen first hand how varying skillsets for players impacts their performance in different positions
- Future players added (like My Player, where you create and choose which skills to level) can have a position prediction made to see where they might fit best on the field, and skill sets can be tailored to a specific position. FIFA developers could also utilize this program to create more accurate skillsets for new players introduced into the game based on their real life positions

The metric used to measure performance is prediction accuracy (i.e. a score of .90 means 90% of the datapoints had the correct class label (position) and 10% had an incorrect class label). This metric may be harsh because it treats all misses the same and doesn't attempt to quantify how bad a miss truly is (i.e. labeling a CM a GK is worse than labeling a CM a RM since RM is a slight variation on the CM position). Regardless, since we only care about the true position of the player, prediction accuracy is the best measure of performance.

Previewing the distribution of data points using a 2D and 3D transformation we can see it is a non-trivial distribution. (Note the one outlier cluster, this is most likely the GK class, which logically has a distinct set of skills for soccer seperate from the other positions)

In [12]:
```
plt.rcParams['figure.figsize'] = [12, 12]
plot_2D_proj(X,b) # Applying PCA and projecting in 2D
print('')
```

In [13]:
```
plot_3D_proj(X,b) # Applying PCA and projecting in 2D
print('')
```



Description of the Algorithms

There were three algorithms applied to the data:

1. Support Vector Machines
2. Random Forests
3. Neural Networks

Support Vector Machines

Description: A classifier that classifies data by creating a hyperplane (think of a line in 2D or plane in 3D) that seperates different classes in the data. The hyperplane also has support vectors, which are boundaries on either side of the hyperplane with the closest data points to the hyperplane on them ("edge cases" for each class are on the support vector). The hyperplane is then calculated from the datapoints from each class that are on the support vectors. SVMs also operate on the large-margin principle, which means the optimal hyperplane is the hyperplane with the largest distance between the support vectors.

Hyperparameters:

- *kernel:* Transformation to be applied to the data before determining the hyperplane. Useful for transforming non-linearly seperable data into a higher dimension where the data is linearly seperable.
- *C:* Regularization parameter that controls the trade off between misclassification and margin size. Larger C values have less misclassification and smaller margin sizes, and thus are more complex.
- *gamma:* Controls how many points are considered when determining the hyperplane. Larger gamma values only include points on or extremely close to the support vectors, which results in decreased complexity since less points are considered.

Random Forests

Description: a classifier that is actually a collection of decision trees, where each decision tree is built from random splits of observations and features in the data. Each decision tree "votes" on what the class label of a point should be. The predicted class label of a new observation is then the class label with the most votes in the random forest (most votes by all the decision trees).

Hyperparameters:

- *bagging vs boosting:* Bagging uses random partitioning of features and observations to create every decision tree randomly from the dataset. This is less complex than its counterpart, boosting. Boosting uses random samples to create trees, but the performance of every tree influences the next tree fit. Thus, each consecutive tree should have a better performance than the last. This leads to increased complexity since each fit tree has to be tested.
- *n_estimators:* Number of decision trees to create for the random forest. More decision trees leads to higher complexity
- *max_features:* The max number of features that can be used to split a node. Larger values lead to increased complexity

Neural Networks

Description: A classifier built on the idea of human brain neural networks. Weighted combinations of inputs (signals) are passed into a perceptron, and each perceptron contains an activation function. If the output of the activation function is above some threshold, the weighted inputs are propogated to the next perceptron (this is called forward propogation). This process is continued through each perceptron until an output is reached. The performance of the model is tested and then the error is sent backwards where the weights used on the inputs of each perceptron are updated to improve model performance (this is backpropogation). This process is repeated over the number of epochs to iteratively improve performance of the neural network.

Hyperparameters:

- *# of hidden layers:* The number of hidden layers of perceptrons to include in the model. More hidden layers leads to increased complexity since more weights need to be updated during forward propogation and backpropogation
- *# of nodes in each hidden layer:* The number of perceptrons to include in each hidden layer. More perceptrons leads to increased complexity since more weights need to be updated during backpropogation and the activation function needs to be run for each perceptron.
- *activation function:* The function used to determine if weighted inputs to a perceptron should be forward propogated to the next perceptron
- *learning rate:* The learning rate to be used during stochastic gradient descent. Smaller learning rates lead to increased complexity since it will take a higher number of epochs to reach the minima of the loss function
- *number of epochs:* The number of iterations of forward propogation and backpropogation training to perform on the data. More epochs gives the neural network more time to adjust weights and fit the training data but also leads to increased complexity

Tuning the Hyperparameters

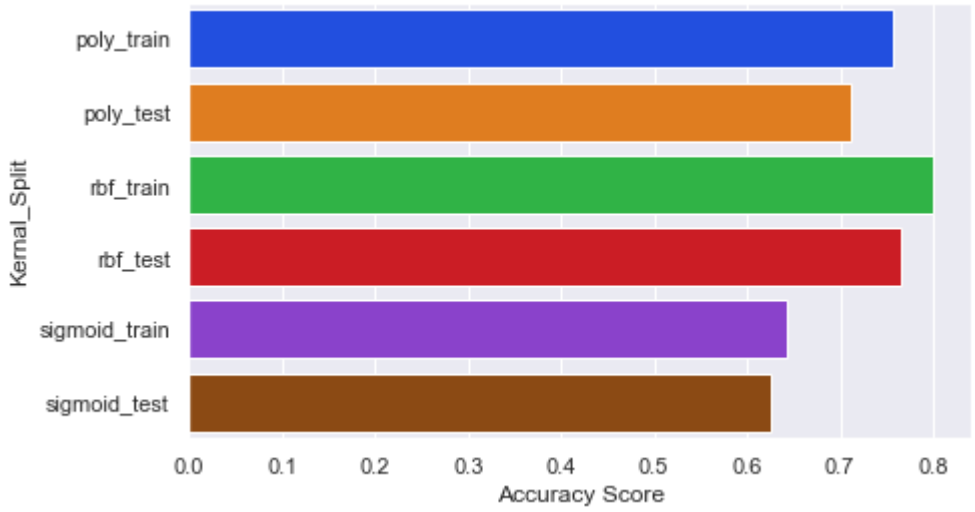
In [14]:
```
plt.rcParams['figure.figsize'] = [7, 4]
X = standardize(X) # normalize data
X, Xho, b, bho = create_test_train_split(X, b, test_size=0.33, random=False) #
create holdout sets
Xcopy = X # copy of X before any transformations are made
```

Support Vector Machines

Testing performance for 'poly', 'rbf', and 'sigmoid' kernels with C=1.0 and gamma='auto'

In [15]:
```
start=time.time()
svm_helper(C=1.0, gamma='auto')
print('experiment runtime: {}'.format(time.time()-start))
```

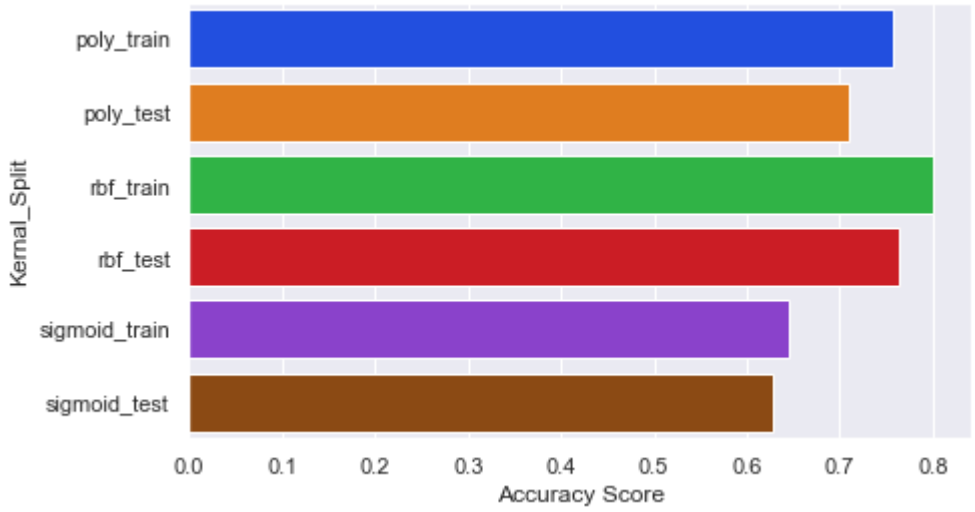
```
{'poly_test': 0.7110754414125201,
 'poly_train': 0.757709947809584,
 'rbf_test': 0.7646869983948635,
 'rbf_train': 0.8012019610944172,
 'sigmoid_test': 0.6263242375601926,
 'sigmoid_train': 0.6433654910643681}
experiment runtime: 18.755135536193848
```



Testing performance for 'poly', 'rbf', and 'sigmoid' kernels with C=1.0 and gamma='scale'

In [16]:
```
start = time.time()
svm_helper(C=1.0, gamma='scale')
print('experiment runtime: {}'.format(time.time()-start))
```

```
{'poly_test': 0.7107544141252007,
 'poly_train': 0.7580262533607465,
 'rbf_test': 0.7646869983948635,
 'rbf_train': 0.8013601138699984,
 'sigmoid_test': 0.6272873194221509,
 'sigmoid_train': 0.6446307132690179}
experiment runtime: 17.6597261428833
```



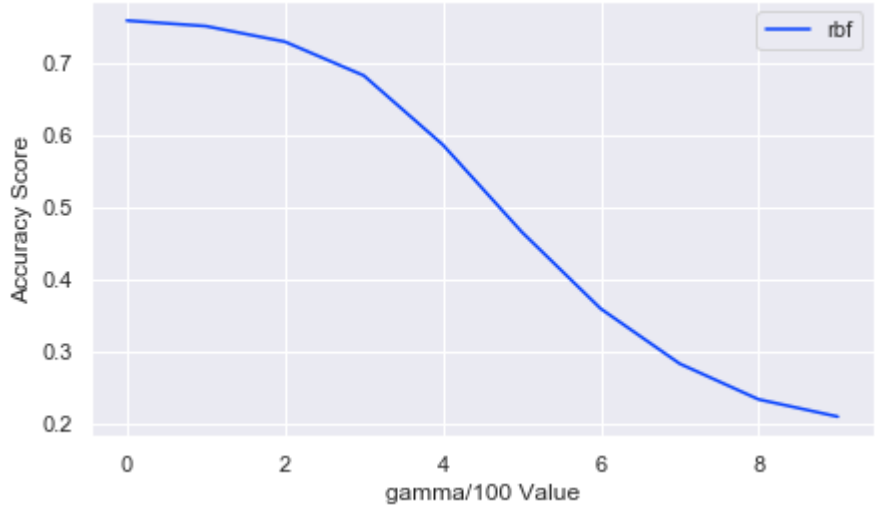
'rbf' is clearly the best kernal for this dataset with the best performance regardless of gamma. Next we test different values for C and gamma with the 'rbf' kernel on this dataset.

In [17]:
```
start = time.time()
perf = []
for i in range(1,100,10):
    results = svm_helper(C=i/100, gamma='auto', printq=False)
    perf.append(results['rbf_test'])
sns.lineplot(data=pd.DataFrame.from_dict({'rbf':perf})).set_xlabel('C/100 Valu
e')
plt.ylabel('Accuracy Score')
print('experiment runtime: {}'.format(time.time()-start))
```

```
experiment runtime: 228.1852090358734
```

In [18]:
```
start = time.time()
perf = []
for i in range(1,100,10):
    results = svm_helper(C=1.0, gamma=i/100, printq=False)
    perf.append(results['rbf_test'])
sns.lineplot(data=pd.DataFrame.from_dict({'rbf':perf})).set_xlabel('gamma/100
 Value')
plt.ylabel('Accuracy Score')
print('experiment runtime: {}'.format(time.time()-start))
```

```
experiment runtime: 352.4892716407776
```



Optimal SVM Model:

From the SVM experiments we see that the rbf kernel consistently outperforms the sigmoid and polynomial (degree=3) kernel. We also see that C values less than 1 decrease test performance of the model. We also see that the model performs best with low values for gamma.

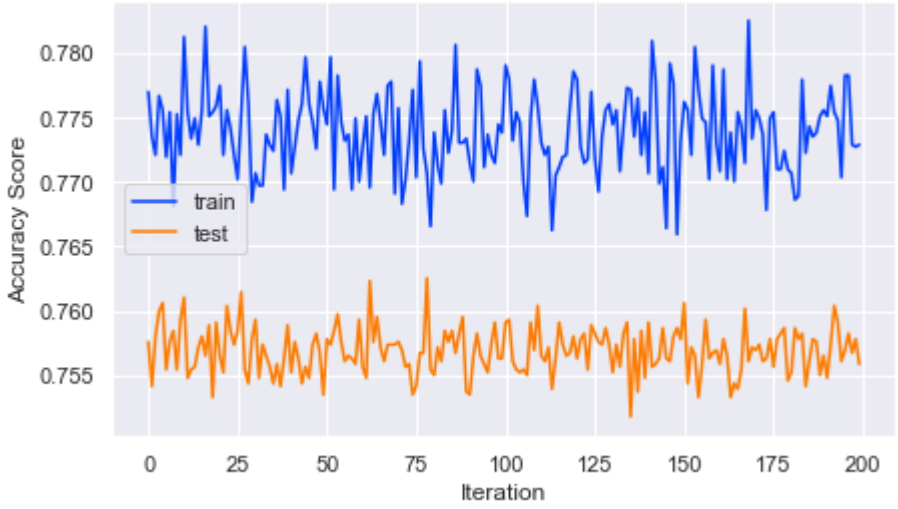
The final hyperparameters chosen for the model are:

- *kernel:* 'rbf'
- *C:* 1.0
- *gamma: 0.01*

Testing the average performance of the model with cross validation we see:

In [19]:
```
expSt = time.time()
train=[]
test=[]
time1=[]
for i in range(200):
    if i % 50 == 0:
        print(int(i/2),'%\t...\t',end='')
    start = time.time()
    svm = SVM(X,b,random=True)
    svm.create_fit(kernel='rbf', c=1.0, gamma=0.01)
    train.append(svm.calc_performance(svm.trainX,svm.trainb))
    test.append(svm.calc_performance(Xho,bho))
    time1.append(time.time()-start)
results1 = pd.DataFrame.from_dict({'train':train,'test':test})
sns.lineplot(data=results1, dashes=False).set_xlabel('Iteration')
plt.ylabel('Accuracy Score')
print('\nTotal experiment runtime: {}s'.format(time.time()-expSt))
```

```
0 %     ...     25 %    ...     50 %    ...     75 %    ...
Total experiment runtime: 1379.2089312076569s
```



In [20]:
```
mean = np.mean(test)
ci = st.t.interval(0.95, len(test)-1, loc=np.mean(test), scale=st.sem(test))
print('avg test accuracy: {}'.format(mean))
print('95% confidence interval on average accuracy: \n\t[{},{}]'.format(ci[0],
ci[1]))
print('avg runtime (per iteration): {}s'.format(np.array(time1).mean()))
```

```
avg test accuracy: 0.756912239191224
95% confidence interval on average accuracy:
        [0.7566569726670413,0.7571675057154066]
avg runtime (per iteration): 6.89495104432106s
```

It's worth noting that despite the increased complexity of *C* and *gamma*, they are still chosen because they have the best model performance. Overall, the model still generalizes extremely well to unseen testing data with less than 0.02 drop in train vs. test performance even with the complexity. That, paired with a relatively fast runtime, makes it worthwhile to choose the higher performance and more complex model.

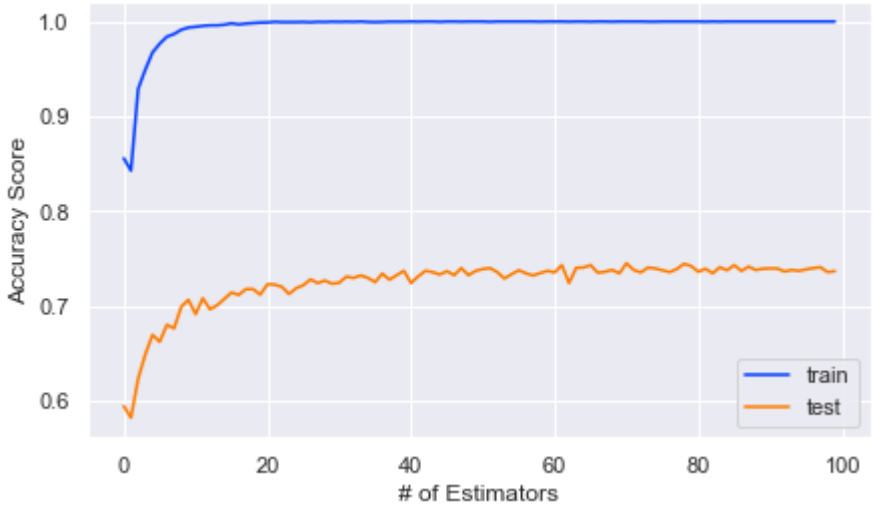
Random Forests

Next we compare random forest performance as we alter bagging vs boosting, max_features, and number of estimators

Fitting a random forest model with bagging and max_features = 6

In [21]:
```
start = time.time()
rf_helper(bag=True, max_features=6)
print('experiment runtime: {}'.format(time.time()-start))
```

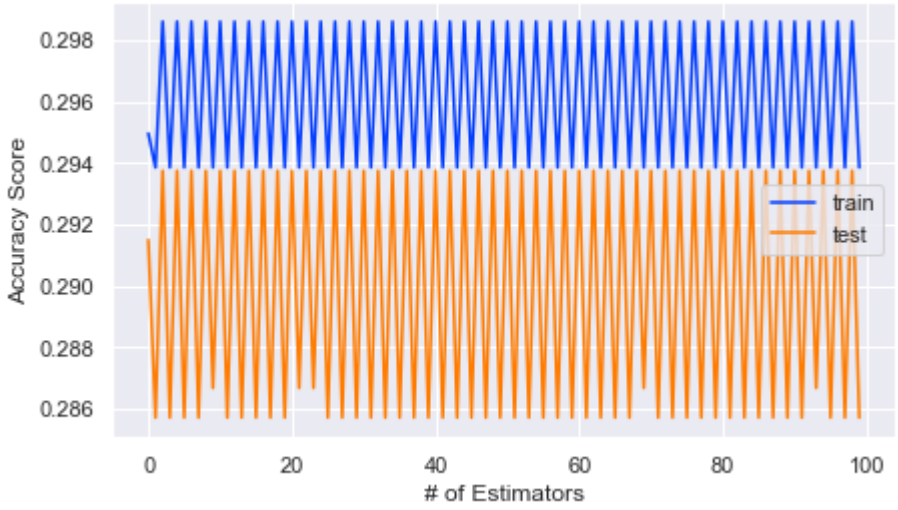
```
Max test score: 0.7451043338683788
experiment runtime: 190.03438806533813
```



Fitting a random forest model with boosting and max_features = 6

In [22]:
```
start = time.time()
rf_helper(bag=False, max_features=6) #boosting
print('experiment runtime: {}'.format(time.time()-start))
```

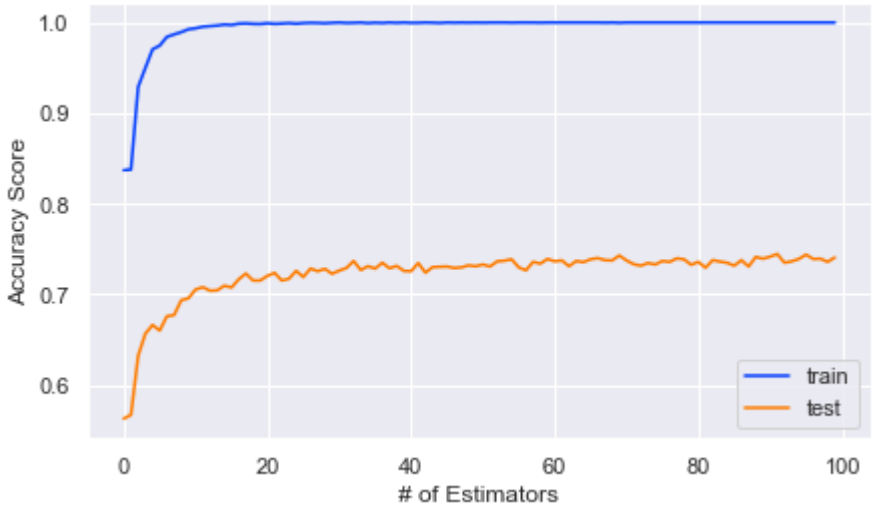
```
Max test score: 0.29373996789727125
experiment runtime: 155.00030517578125
```



Fitting a random forest model with bagging and max_features = 5

In [23]:
```
start = time.time()
rf_helper(bag=True, max_features=5)
print('experiment runtime: {}'.format(time.time()-start))
```

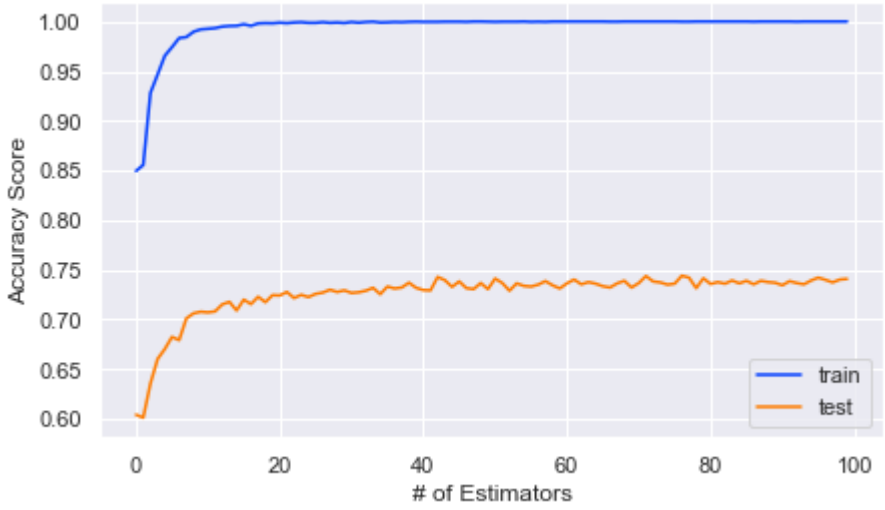
```
Max test score: 0.74446227929374
experiment runtime: 170.90978455543518
```



Fitting a random forest model with bagging using max_features = 17

In [24]:
```
start = time.time()
rf_helper(bag=True, max_features=17)
print('experiment runtime: {}'.format(time.time()-start))
```

```
Max test score: 0.7441412520064206
experiment runtime: 434.8330216407776
```



Optimal Random Forest Model

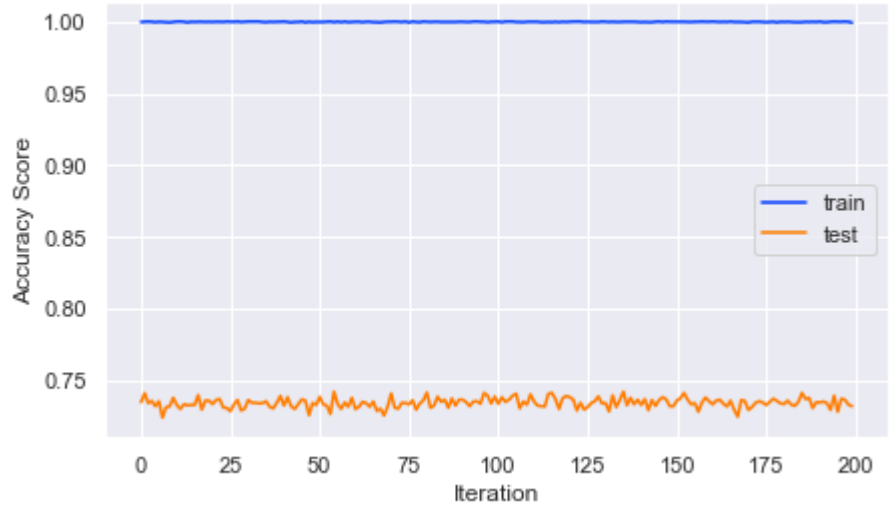
From the RF experiments we see a drastic decrease in model performance when boosting is utilized. Thus, the optimal model utilizes bagging. The best performance from max_features came when 17 features were used, out performing all lower values for max_features. This is slightly higher complexity than the other lower value max_features models but still relatively low complexity overall. The best performance with the lowest n_estimators occured at 43, meaning 43 decision trees were in the random forest.

The final hyperparameters are:

- *bagging vs boosting:* Bagging
- *n_estimators:* 43
- *max_features:* 5

In [25]:
```
expSt = time.time()
train=[]
test=[]
time2=[]
for i in range(200):
    if i % 50 == 0:
        print(int(i/2),'%\t...\t',end='')
    start = time.time()
    rf = RandomForest(X,b,random=True)
    rf.create_fit(bag=True, max_features=17, n_estimators=43)
    train.append(rf.calc_performance(rf.trainX,rf.trainb))
    test.append(rf.calc_performance(Xho,bho))
    time2.append(time.time()-start)
results2 = pd.DataFrame.from_dict({'train':train,'test':test})
sns.lineplot(data=results2, dashes=False).set_xlabel('Iteration')
plt.ylabel('Accuracy Score')
print('\nTotal experiment runtime: {}s'.format(time.time()-expSt))
```

```
0 %     ...     25 %    ...     50 %    ...     75 %    ...
Total experiment runtime: 744.4076113700867s
```



In [26]:
```
mean = np.mean(test)
ci = st.t.interval(0.95, len(test)-1, loc=np.mean(test), scale=st.sem(test))
print('avg test accuracy: {}'.format(mean))
print('95% confidence interval on average accuracy: \n\t[{},{}]'.format(ci[0],
ci[1]))
print('avg runtime (per iteration): {}s'.format(np.array(time2).mean()))
```

```
avg test accuracy: 0.7344826844482685
95% confidence interval on average accuracy:
        [0.734000877384033,0.734964491512504]
avg runtime (per iteration): 3.720984146595001s
```

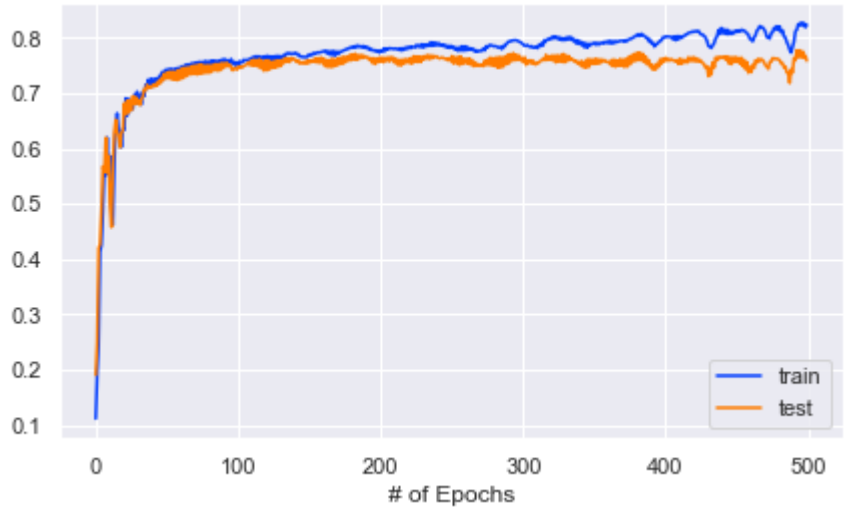
Neural Networks

Finally, we test different neural networks to choose an optimal neural network model

Neural network with 2 hidden layers, 150 perceptrons each, activation_function=relu, lr=0.75, epochs=500

In [27]:
```
start = time.time()
net = Net(X, b, n_hidden=[150,150], n_output=20, activ_func='relu')
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.75)
loss_func = torch.nn.CrossEntropyLoss()
results, runtime, out = net.train_net(num_epochs=500, print_iters=False)
print('Max test score: {}'.format(results['test'].max()))
sns.lineplot(data=results, dashes=False).set_xlabel('# of Epochs')
print('experiment runtime: {}'.format(time.time()-start))
```

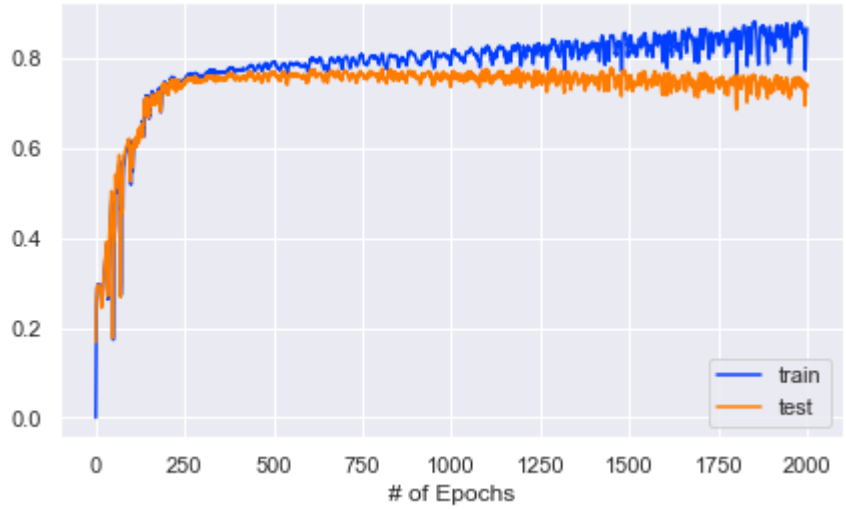
```
Net(
  (hidden1): Linear(in_features=35, out_features=150, bias=True)
  (hidden2): Linear(in_features=150, out_features=150, bias=True)
  (out): Linear(in_features=150, out_features=20, bias=True)
)
Max test score: 0.777207062600321
experiment runtime: 59.41456890106201
```



Neural network with 4 hidden layers, 75 perceptrons each, activation_function=relu, lr=0.3, epochs=2000

In [28]:
```
start = time.time()
net = Net(X, b, n_hidden=[75,75,75,75], n_output=20, activ_func='relu')
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.3)
loss_func = torch.nn.CrossEntropyLoss()
results, runtime, out = net.train_net(num_epochs=2000, print_iters=False)
print('Max test score: {}'.format(results['test'].max()))
sns.lineplot(data=results, dashes=False).set_xlabel('# of Epochs')
print('experiment runtime: {}'.format(time.time()-start))
```

```
Net(
  (hidden1): Linear(in_features=35, out_features=75, bias=True)
  (hidden2): Linear(in_features=75, out_features=75, bias=True)
  (hidden3): Linear(in_features=75, out_features=75, bias=True)
  (hidden4): Linear(in_features=75, out_features=75, bias=True)
  (out): Linear(in_features=75, out_features=20, bias=True)
)
Max test score: 0.7765650080256822
experiment runtime: 224.4491879940033
```

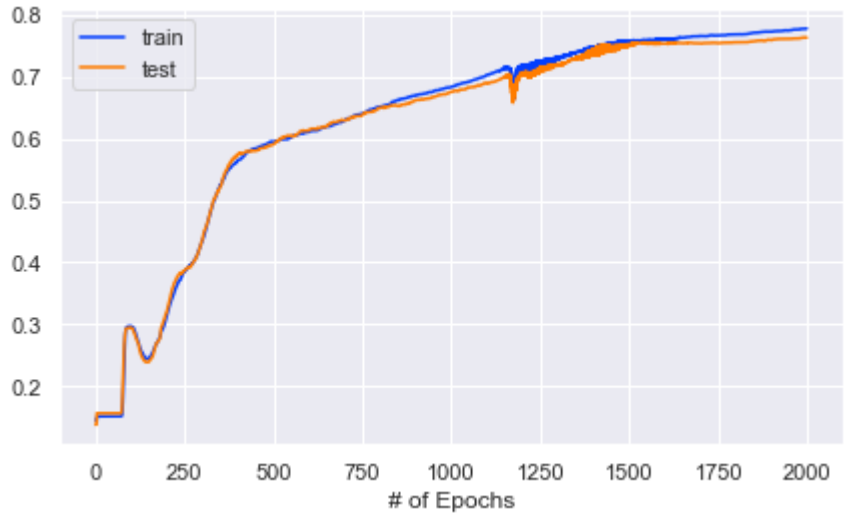


Neural network with 3 hidden layers, 35,55,27 perceptrons respectively, activation_function=sigmoid, lr=0.7, epochs=2000

In [29]:
```
start = time.time()
net = Net(X, b, n_hidden=[35,55,27], n_output=20, activ_func='sigmoid')
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.7)
loss_func = torch.nn.CrossEntropyLoss()
results, runtime, out = net.train_net(num_epochs=2000, print_iters=False)
print('\nmax test score: {}'.format(results['test'].max()))
sns.lineplot(data=results, dashes=False).set_xlabel('# of Epochs')
print('experiment runtime: {}'.format(time.time()-start))
```

```
Net(
  (hidden1): Linear(in_features=35, out_features=35, bias=True)
  (hidden2): Linear(in_features=35, out_features=55, bias=True)
  (hidden3): Linear(in_features=55, out_features=27, bias=True)
  (out): Linear(in_features=27, out_features=20, bias=True)
)

max test score: 0.762760834670947
experiment runtime: 86.77672672271729
```

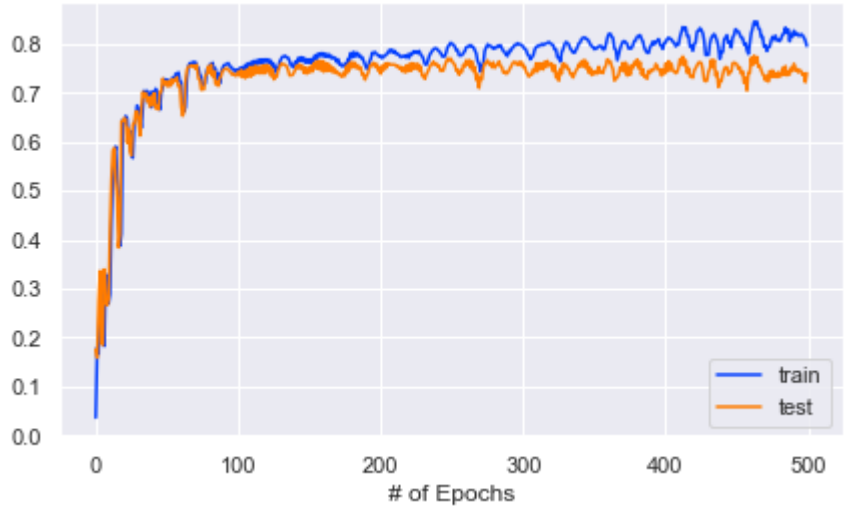


Neural network with 3 hidden layers, 300 perceptrons each, activation_function=relu, lr=0.75, epochs=500

In [30]:
```
start = time.time()
net = Net(X, b, n_hidden=[300,300,300], n_output=20, activ_func='relu')
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.75)
loss_func = torch.nn.CrossEntropyLoss()
results, runtime, out = net.train_net(num_epochs=500, print_iters=False)
print('\nmax test score: {}'.format(results['test'].max()))
sns.lineplot(data=results, dashes=False).set_xlabel('# of Epochs')
print('experiment runtime: {}'.format(time.time()-start))
```

```
Net(
  (hidden1): Linear(in_features=35, out_features=300, bias=True)
  (hidden2): Linear(in_features=300, out_features=300, bias=True)
  (hidden3): Linear(in_features=300, out_features=300, bias=True)
  (out): Linear(in_features=300, out_features=20, bias=True)
)

max test score: 0.7752808988764045
experiment runtime: 186.07071042060852
```

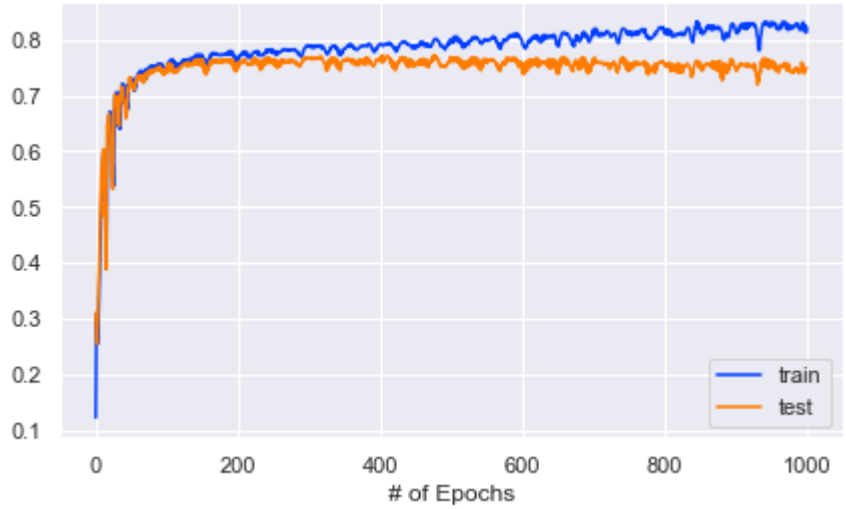


Neural network with 2 hidden layers, 150 perceptrons each, activation_function=relu, lr=0.8, epochs=500

In [31]:
```
start = time.time()
net = Net(X, b, n_hidden=[50,50], n_output=20, activ_func='relu')
print(net)
optimizer = torch.optim.SGD(net.parameters(), lr=0.8)
loss_func = torch.nn.CrossEntropyLoss()
results, runtime, out = net.train_net(num_epochs=1000, print_iters=False)
print('\nmax test score: {}'.format(results['test'].max()))
sns.lineplot(data=results, dashes=False).set_xlabel('# of Epochs')
print('experiment runtime: {}'.format(time.time()-start))
```

```
Net(
  (hidden1): Linear(in_features=35, out_features=50, bias=True)
  (hidden2): Linear(in_features=50, out_features=50, bias=True)
  (out): Linear(in_features=50, out_features=20, bias=True)
)

max test score: 0.7717495987158909
experiment runtime: 45.39843201637268
```



Optimal Neural Network Model

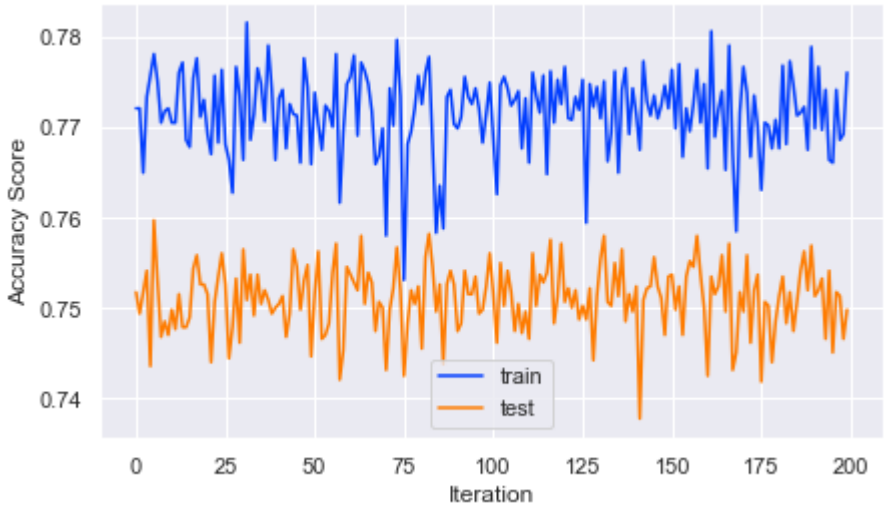
First, we note that the model with the 2nd best performance had 2 hidden layers with 150 nodes each. Decreasing the number of nodes per hidden layer from 150 to 50 increased the model performance even further on the testing data (due to decreased complexity and thus better generalization). Also note that the other, more complex models with more hidden layers and more nodes in the hidden layers did not outperform the less complex neural network. From the NN experiments we can also see that most of the models converged on the training data by 500 epochs, and afterwards we see overfitting to training data that does not improve the test performance. For complexity, runtime, and performance we choose 200 for the number of epochs. Also, note that lowering the learning rate did not improve model performance and mainly just took longer to find the optimal weights, so we choose a learning rate of 0.8. Lastly, it is clear that the 'relu' activation function outperformed the 'sigmoid' activation function.

The final hyperparameters are:

- *# of hidden layers:* 2
- *# of nodes in each hidden layer:* 50
- *activation function:* 'relu'
- *learning rate:* 0.8
- *number of epochs:* 200

In [32]:
```
expSt = time.time()
train=[]
test=[]
time3=[]
for i in range(200):
    if i % 50 == 0:
        print(int(i/2),'%\t...\t',end='')
    start = time.time()
    net = Net(X, b, n_hidden=[50,50], n_output=20, activ_func='relu')
    optimizer = torch.optim.SGD(net.parameters(), lr=0.8)
    loss_func = torch.nn.CrossEntropyLoss()
    results, runtime, out = net.train_net(num_epochs=200)
    train.append(results['train'][len(results['train'])-1])
    test.append(net.test_net(test_final=True))
    time3.append(time.time()-start)
results3 = pd.DataFrame.from_dict({'train':train,'test':test})
sns.lineplot(data=results3, dashes=False).set_xlabel('Iteration')
plt.ylabel('Accuracy Score')
print('\nTotal experiment runtime: {}s'.format(time.time()-expSt))
```

```
0 %     ...     25 %    ...     50 %    ...     75 %    ...
Total experiment runtime: 1703.612209558487s
```



In [33]:
```
mean = np.mean(test)
ci = st.t.interval(0.95, len(test)-1, loc=np.mean(test), scale=st.sem(test))
print('avg test accuracy: {}'.format(mean))
print('95% confidence interval on average accuracy: \n\t[{},{}]'.format(ci[0],
ci[1]))
print('avg runtime (per iteration): {}s'.format(np.array(time3).mean()))
```

```
avg test accuracy: 0.7510755001075501
95% confidence interval on average accuracy:
        [0.7505563268966702,0.75159467331843]
avg runtime (per iteration): 8.51698388338089s
```

Comparing Algorithm Performance

To compare each algorithms respective performance, 200 iterations of training with random train splits and testing on held out data (that was not seen during hyper parameter tuning) was performed for each algorithm. The performance of each algorithm was recorded and plotted below.

Comparing only performance on the held out testing data (validation performance)

In [34]:
```
fig = plt.figure()
ax = plt.subplot(111)
results={}
results['nn_test'] = results3['test']
results['rf_test'] = results2['test']
results['svm_test'] = results1['test']
results = pd.DataFrame.from_dict(results)
sns.lineplot(data=results, dashes=False).set_xlabel('Iteration')
ax.legend(loc='upper center', bbox_to_anchor=(1.15, 0.8), shadow=False, ncol=1
)
plt.ylabel('Validation Accuracy Score')
print('Validation Accuracy vs. Iteration')
```

```
Validation Accuracy vs. Iteration
```

From the plot above we can see that the SVM model consistently outperformed the other 2 algorithms (the neural network model outperformed the SVM model once). We also note that SVM model was the most consistent in validation accuracy (had the least variance) of all the models.

From the SVM optimal model, I am 95% confident that the SVM model will average correctly classifying between 0.756% and 0.757% of points in any batch shown to it.

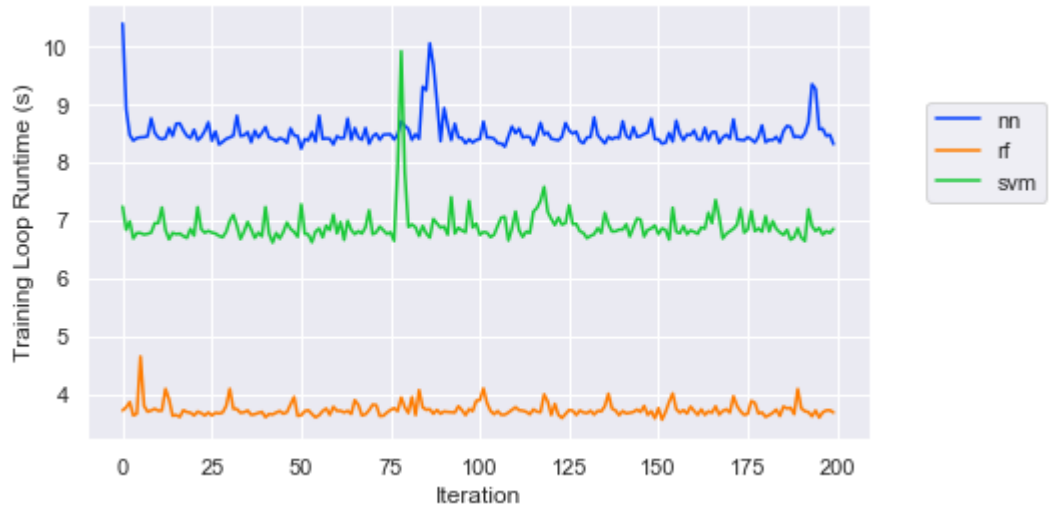
For the random forest model, I am 95% confident that the average validation score is between 0.7335% and 0.7345% for any future batches presented to the model.

For the neural network model, I am 95% confident that the average validation score is between 0.7506% and 0.7516% for any future batches presented to the model.

Comparing runtimes for each algorithm:

In [35]:
```
fig = plt.figure()
ax = plt.subplot(111)
results={}
results['nn'] = time3
results['rf'] = time2
results['svm'] = time1
results = pd.DataFrame.from_dict(results)
sns.lineplot(data=results, dashes=False).set_xlabel('Iteration')
ax.legend(loc='upper center', bbox_to_anchor=(1.15, 0.8), shadow=False, ncol=1
)
plt.ylabel('Training Loop Runtime (s)')
print('Runtime vs. Iteration')
```

```
Runtime vs. Iteration
```



Conclusion

In conclusion, when choosing an algorithm to use for data from this domain, there are 3 important factors.

- Validation Accuracy
- Train Time
- Complexity

Using the validation accuracy and runtime plots provided above alongside the hyperparemeter tuning, we can easily rank the three algorithms in each category

1. **Support Vector Machines:**
    - Validation Acuracy: Best
    - Train Time: Moderate
    - Complexity: Moderate Complexity (3 hyperparameters, tuned to be high complexity)
2. **Random Forests:**
    - Validation Accuracy: Worst
    - Train Time: Best
    - Complexity: Least Complex (3 hyperparameters, tuned to be low complexity)
3. **Neural Networks:**
    - Validation Accuracy: Moderate
    - Train Time: Worst
    - Complexity: Most Complex (5 hyperparameters)

Thus, the algorithm I would utilize on real world data from the same domain would be Support Vector Machines with the 'rbf' kernel, C = 1.0, and gamma = 0.01 . This algorithm outperformed both random forests and neural networks in validation accuracy and took less time to train than neural networks (but more than random forests). Despite *C* and *gamma* being more complex in the model, the difference between train and test accuracy is very small (less than 0.02), so the increased complexity still generalizes well and does not take too long to run.

If a new dataset from the same domain needed to be tested that was significantly larger than the current dataset, I would choose random forests as my algorithm. The fast training time and small complexity would generalize well to a large dataset quickly and provide decent validation accuracy.

The neural network model is very complex and offers no benefit to validation accuracy or training runtime, thus, I would not use neural networks in the real world on data from this domain.

Acknowledgements

PyTorch docs (https://pytorch.org/docs/stable/index.html)

FIFA 19 Dataset on Kaggle (https://www.kaggle.com/karangadiya/fifa19)

SciKit-Learn docs (https://scikit-learn.org/stable/index.html)

Seaborn docs (https://seaborn.pydata.org/index.html)

Additional Research on SVMs (https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72)

Additional Research on Random Forests (https://builtin.com/data-science/random-forest-algorithm#hyperparameters)

Additional Research on Neural Networks (http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414)

Future Research

For those interested, there are still several interesting questions that can be raised from this analysis

- Which positions are misclassified most often?
- Is there a correlation between any one feature and misclassification (i.e. do higher 'Overall' players have more customized feature sets that lead to increased misclassification?)
- Would breaking positions into larger buckets improve performance (i.e attacker, midfield, defensive)

Back to Top

In [36]:
```
print('Notebook Total Runtime: {}'.format(time.time() - bookStart))
```

```
Notebook Total Runtime: 6017.4951910972595
```