In [1]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

%matplotlib inline

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
sns.set_context("notebook")
#sns.set_context("poster")
```

In [2]:

```
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score

from sklearn.metrics import accuracy_score

from sklearn import preprocessing
```

Ensembles

Ensembles develop around two main ideas.

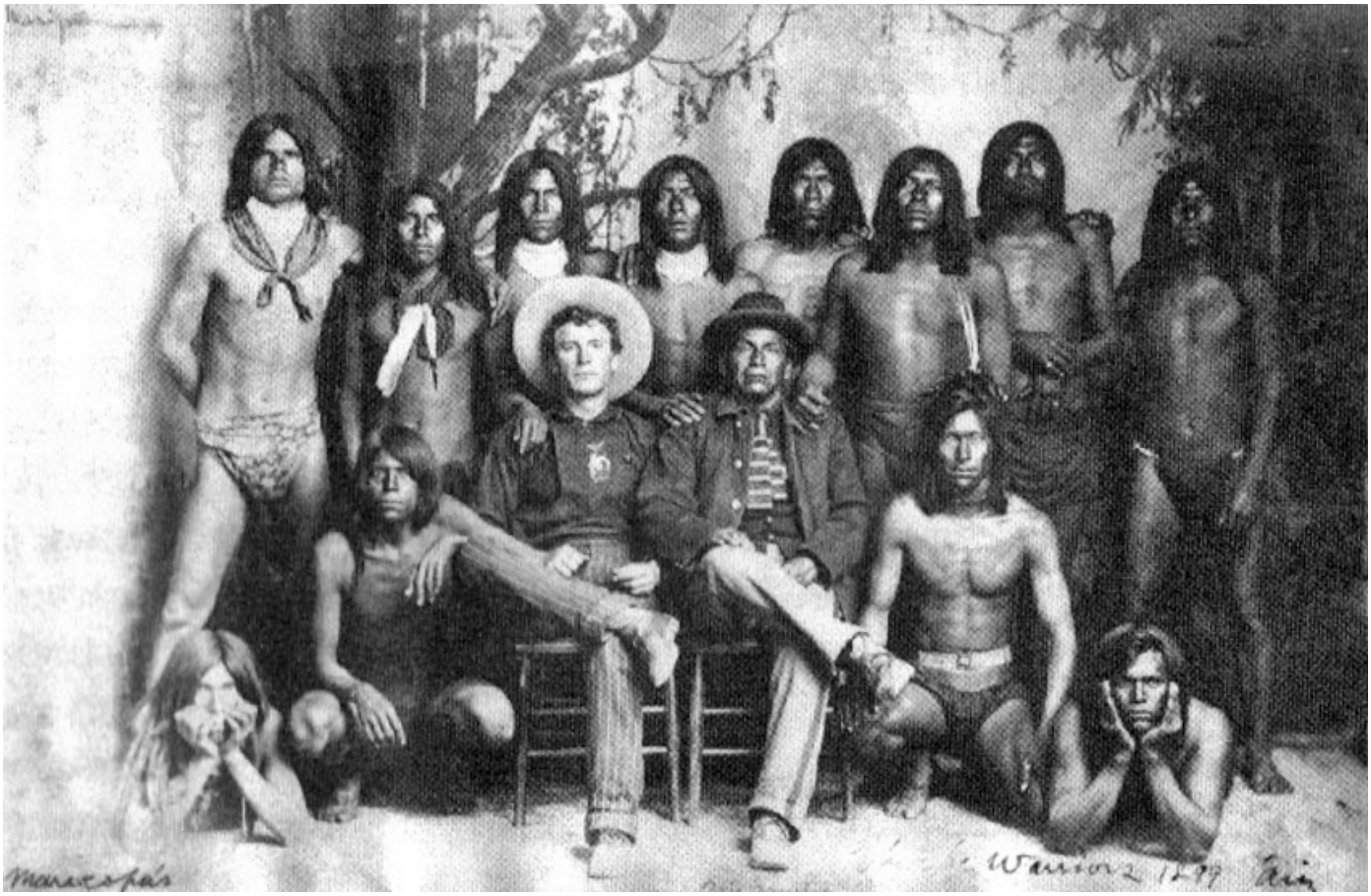
The first one the idea that combining weak learners we can get a strong learner. Around this idea there is a large corpus of theoretical work that gets implemented and refined through time.

The second main idea is more prosaic and revolves around the need to overcome overfitting, particularly in trees. This results in implementing combinations of the same learner in order to reduce variance and avoid overfitting while increasing the performance of the learner.

These ideas crystalize in three different models of ensembles:

- **Bagging.** Building multiple models, typically the same type, from different subsamples of a dataset (normmally with repetition) and combining them with an aggregate such as the mean.
- **Boosting.** The idea of boosting is to build the model incrementally where each iteration tries to fix the errors of the previous one.
- **Voting.** In this case we have multiple models, typically of different types, and a procedure to combine their predictions (normlly a simple statistic such as the mean).

In order to be able to compare them with the previous one, we will use the same dataset, the Pima Indians, with a 10-fold cross-validation and accuracy as the performance metric.



In this exercise we will use one of the traditional Machine Learning dataset, the Pima Indians diabetes dataset.

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Content The datasets consists of several medical predictor variables and one target variable, **Outcome**. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

- Pregnancies
- Glucose
- BloodPressure
- SkinThickness
- Insulin
- BMI
- DiabetesPedigreeFunction (scores de likelihood of diabetes based on family history)
- Age
- Outcome

In [3]:

```
# Load the Pima indians dataset and separate input and output components

from numpy import set_printoptions
set_printoptions(precision=3)

filename="pima-indians-diabetes.data.csv"
names=["pregnancies", "glucose", "pressure", "skin", "insulin", "bmi", "pedi", "age", "outc
p_indians = pd.read_csv(filename, names=names)
p_indians.head()

# First we separate into input and output components
array = p_indians.values
X = array[:,0:8]
y = array[:,8]
np.set_printoptions(suppress=True)
X
pd.DataFrame(X).head()

# Create the DataFrames for plotting
resall=pd.DataFrame()
res_w1=pd.DataFrame()
```

Out[3]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Out[3]:

```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Out[3]:

	0	1	2	3	4	5	6	7
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0
3	1.0	89.0	66.0	23.0	94.0	28.1	0.167	21.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0

Bagged Decision Trees

Bagging is the contraction of bootstraping + aggregation. The idea behind bagging is to reduce the variance of the weak learner by randomly sampling with repetition and building a number of learners than later are being aggregated with voting if a classifier or with an statistic such as the mean if regression.

In this case we will use the DecisionTreeClassifier (CART) with the **BaggingClassifier** class.

In [4]:

```
# Bagged Decision Trees

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

seed = 7
kfold = KFold(n_splits=10, random_state=seed, shuffle = True)

#learner=DecisionTreeClassifier(class_weight="balanced", random_state=seed)

learner = DecisionTreeClassifier(random_state = seed)
num_trees = 100
model = BaggingClassifier(base_estimator = learner, n_estimators = num_trees, random_state
results = cross_val_score(model, X, y, cv=kfold)

print(f'Bagged Decision Trees - Accuracy {results.mean()*100:.3f}% std {results.std()*100:3

res_w1["Res"] = results
res_w1["Type"] = "Bagged DT"

resall = pd.concat([resall,res_w1], ignore_index=True)
```

```
Bagged Decision Trees - Accuracy 75.913% std 3.492709
```

Random Forest

Random Forest is an extension of Bagged Decision Trees, aiming at reducing the correlation between the individual classifiers.

The strategy chosen consists in considering a randomly selected number of features in each split instead of searching greedily the best.

For Random Forest you have to use the **RandomForestClassifier** class.

In [5]:

```
# Random Forest

from sklearn.ensemble import RandomForestClassifier

seed = 7
kfold = KFold(n_splits=10, random_state=seed, shuffle = True)

num_trees = 100
num_features = 3

model = RandomForestClassifier(n_estimators = num_trees, max_features = num_features, rando
results = cross_val_score(model, X, y, cv = kfold)

print(f'Random Forest - Accuracy {results.mean()*100:.3f}% std {results.std()*100:3f}')

res_w1["Res"] = results
res_w1["Type"] = "Random Forest"

resall = pd.concat([resall,res_w1], ignore_index=True)
```

```
Random Forest - Accuracy 75.911% std 4.632629
```

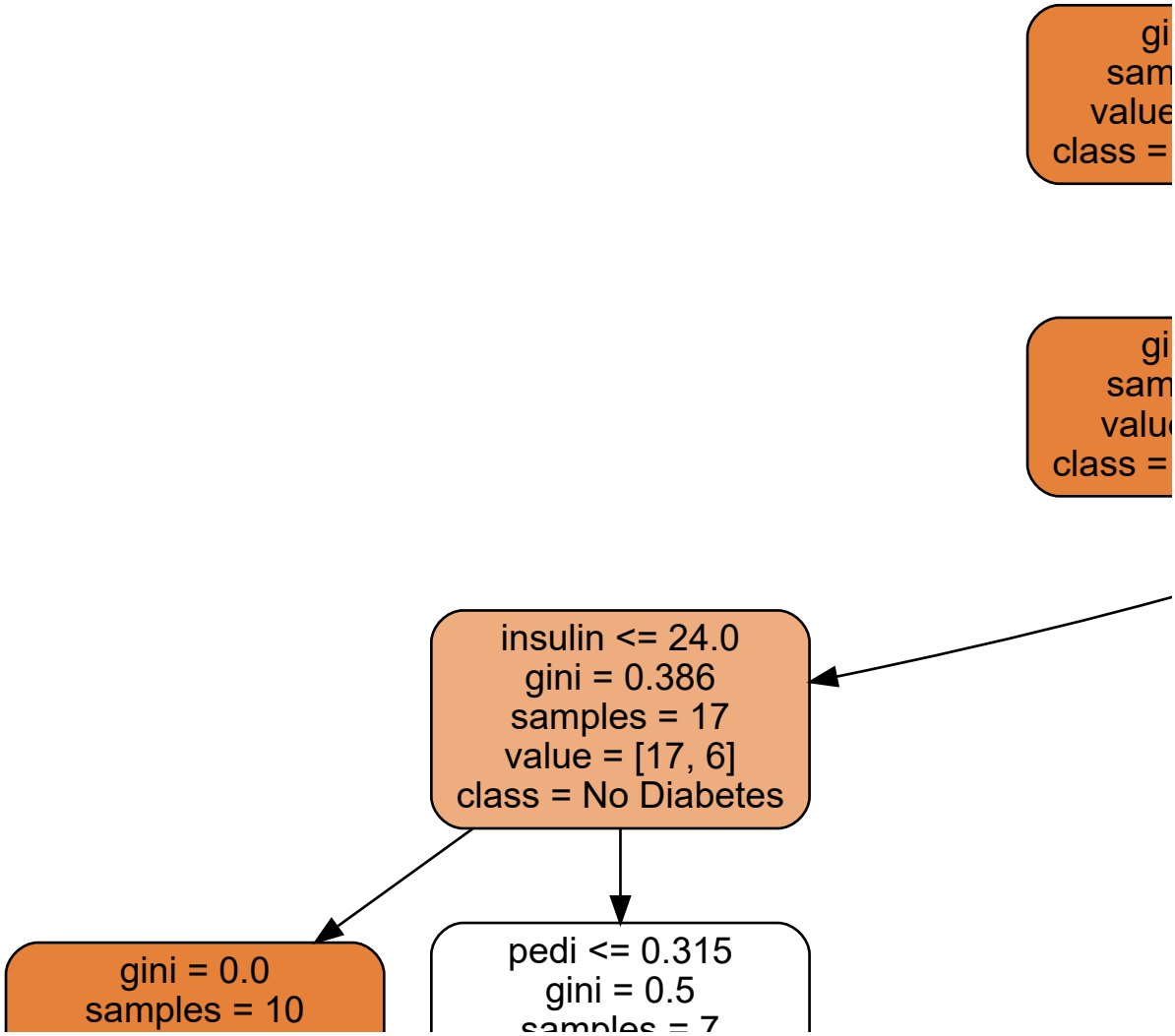
In [6]:

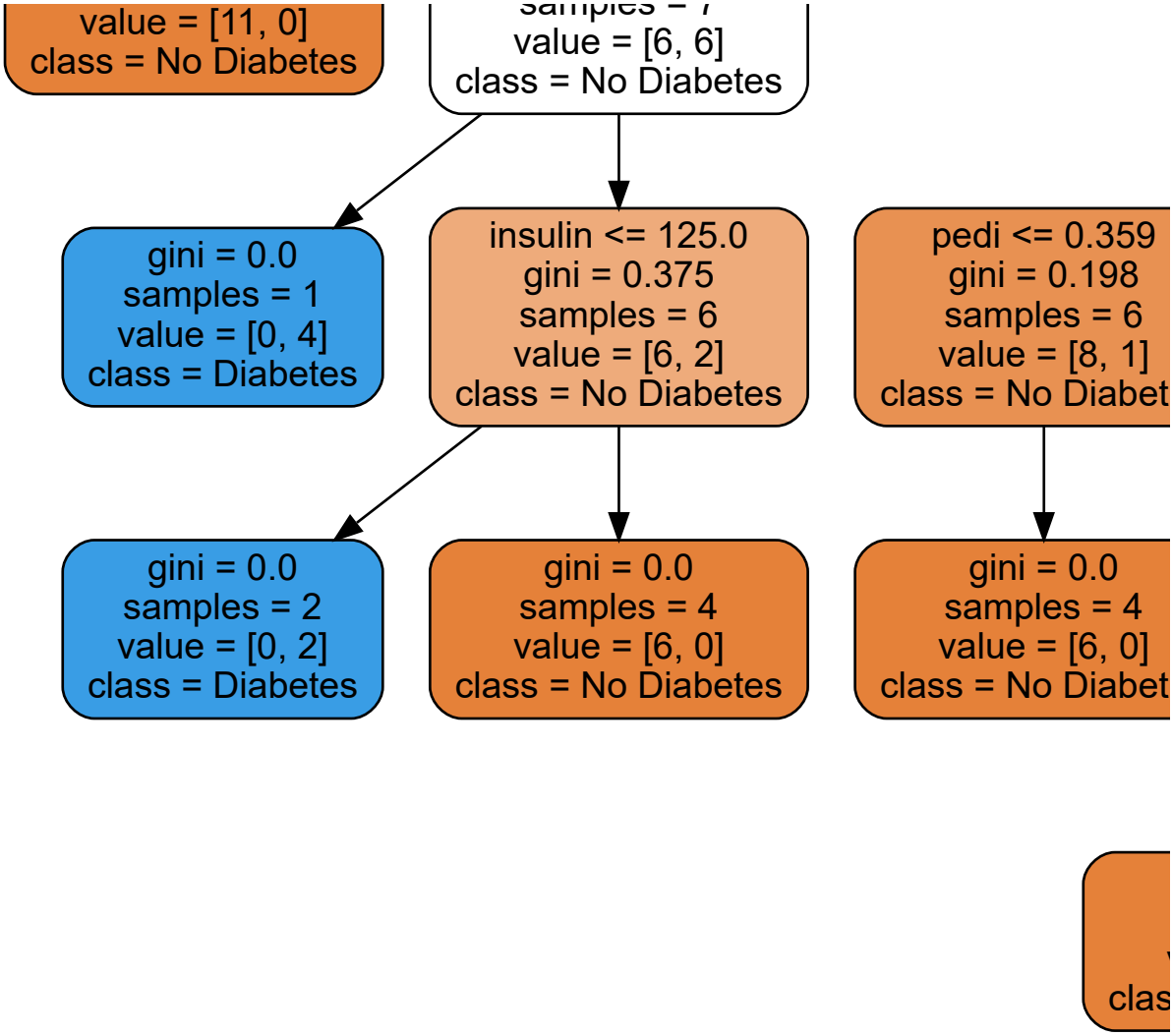
```
# visualizing a single tree in a random forest

from sklearn.ensemble import RandomForestClassifier

from sklearn import tree
from graphviz import Source
from IPython.display import SVG, display
from ipywidgets import interactive

seed = 7

num_trees = 100
num_features = 3

model = RandomForestClassifier(n_estimators = num_trees, max_features = num_features, rando
model.fit(X,y)

estimator = model.estimators_[5]

graph = Source(tree.export_graphviz(estimator,
        out_file=None,
        feature_names=p_indians.columns[:-1],
        class_names=['No Diabetes','Diabetes'],
        filled=True,
        rounded=True))

graph

#if you want to save it in a file
#  the file will open in preview and you can save it
#  just uncomment

#graph.format = 'png'
#graph.render('dtree_render',view=True)
```

Out[6]:

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features=3,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=7, verbose
=0,
                       warm_start=False)
```

Out[6]:


gi
sam
value
class =
gi
sam
valu
class =
insulin <= 24.0
gini = 0.386
samples = 17
value = [17, 6]
class = No Diabetes
gini = 0.0
samples = 10
pedi <= 0.315
gini = 0.5


value = [11, 0]
class = No Diabetes
value = [6, 6]
class = No Diabetes
gini = 0.0
samples = 1
value = [0, 4]
class = Diabetes
insulin <= 125.0
gini = 0.375
samples = 6
value = [6, 2]
class = No Diabetes
pedi <= 0.359
gini = 0.198
samples = 6
value = [8, 1]
class = No Diabet
gini = 0.0
samples = 2
value = [0, 2]
class = Diabetes
gini = 0.0
samples = 4
value = [6, 0]
class = No Diabetes
gini = 0.0
samples = 4
value = [6, 0]
class = No Diabet

Extra Trees

Extra Trees stands for Extremely Randomized Trees and it's a variation of Random Forest.

While similar to ordinary random forests in that they are an ensemble of individual trees, there are two main differences: first, each tree is trained using the whole learning sample (rather than a bootstrap sample), and second, the top-down splitting in the tree learner is randomized. Instead of computing the locally optimal cut-point for each feature under consideration (based on, e.g., information gain or the Gini impurity), a random cut-point is selected.

For Extra Tress you must use the **ExtraTreeClassifier** class.

In [7]:

```
# Extra Trees

from sklearn.ensemble import ExtraTreesClassifier

seed = 7

kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

num_trees = 300
num_features = 5

model = ExtraTreesClassifier(n_estimators = num_trees, max_features = num_features, random_

results = cross_val_score(model, X, y, cv = kfold)

print(f'Extra Trees - Accuracy {results.mean()*100:.3f}% std {results.std()*100:3f}')

res_w1["Res"] = results
res_w1["Type"] = "Extra Trees"

resall = pd.concat([resall,res_w1], ignore_index=True)
```

```
Extra Trees - Accuracy 76.951% std 4.622462
```

AdaBoost

AdaBoost, short for Adaptative Boosting, was the first really successful boosting algorithm and in many ways opened the way to a new generation of boosting algorithms.

It works by weighting instances of the dataset according to their difficulty to classify and using these weights to pay more or less attention to each instance when constructing the subsequent models.

You can use AdaBoost for classification with the **AdaBoostClassifier** class.

In [8]:

```
# AdaBoost

from sklearn.ensemble import AdaBoostClassifier

seed=7

kfold=KFold(n_splits=10, random_state=seed, shuffle = True)

num_trees=30

model=AdaBoostClassifier(n_estimators=num_trees, random_state=seed)

results=cross_val_score(model, X, y, cv=kfold)

print(f'AdaBoost - Accuracy {results.mean()*100:.3f}% std {results.std()*100:3f}')

res_w1["Res"]=results
res_w1["Type"]="AdaBoost"

resall=pd.concat([resall,res_w1], ignore_index=True)
```

```
AdaBoost - Accuracy 75.528% std 3.714314
```

Stochastic Gradient Boosting

Stochastic Gradient Boosting (also called Gradient Boosting Machines) is one of the most sophisticated ensemble techniques and one of the best in terms of improving the performance of ensembles.

For Stochastic Gradient Boosting you have to use the **GradientBoostingClassifier** class.

In [9]:

```
# Stochastic Gradient Boosting

from sklearn.ensemble import GradientBoostingClassifier

seed=7
num_trees=30

model=GradientBoostingClassifier(n_estimators=num_trees, random_state=seed)

results=cross_val_score(model, X, y, cv=kfold)

print(f'Stochastic Gradient Boosting - Accuracy {results.mean()*100:.3f}% std {results.std(

res_w1["Res"]=results
res_w1["Type"]="GradientBoosting"

resall=pd.concat([resall,res_w1], ignore_index=True)
kfold=KFold(n_splits=10, random_state=seed, shuffle = True)
```

```
Stochastic Gradient Boosting - Accuracy 76.048% std 3.827065
```

Voting Ensemble

Voting is the simplest way to aggregate the predictions of multiple classifiers.

The idea behind is pretty straighforward. First you create all models using your training dataset and when predicting you average (or vote in case of a classifier) the predictions of the submodels.

More evolved variations can learn automatically how to best weight the predictions from the sub-models, although these versions are not currently available in scikit-learn

You can create a voting ensemble with the **VotingClassifier** class.

In [10]:

```
# Voting Ensemble

from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

seed = 7

kfold = KFold(n_splits=10, random_state=seed, shuffle = True)

# create the models
estimators = []
model1 = LogisticRegression(solver="liblinear")
estimators.append(("logistic", model1))

model2 = DecisionTreeClassifier(random_state=seed)
estimators.append(("cart", model2))

model3=SVC(gamma="auto")
estimators.append(("svm", model3))

num_trees = 100
num_features = 3

model4 = RandomForestClassifier(n_estimators=num_trees, max_features=num_features, random_s
estimators.append(("rfc", model4))

model = VotingClassifier(estimators)

results = cross_val_score(model, X, y, cv=kfold)

print(f'Voting Ensemble (log,cart,rfc) - Accuracy {results.mean()*100:.3f}% std {results.st

res_w1["Res"]=results
res_w1["Type"]="Voting"

resall = pd.concat([resall,res_w1], ignore_index=True)
```

```
Voting Ensemble (log,cart,rfc) - Accuracy 74.879% std 4.382285
```

Feature Importance

In [11]:

```
# Random Forest

plt.figure(figsize=(15,9))

from sklearn.ensemble import RandomForestClassifier

seed = 7

num_trees = 100
num_features = 3

model = RandomForestClassifier(n_estimators = num_trees, max_features = num_features, rando
model.fit(X,y)

for name, importance in zip(p_indians.columns, model.feature_importances_):
    print(f'{name:15s}  {importance:.4f}')

sns.barplot(x = p_indians.columns[:-1], y = model.feature_importances_)
```

Out[11]:

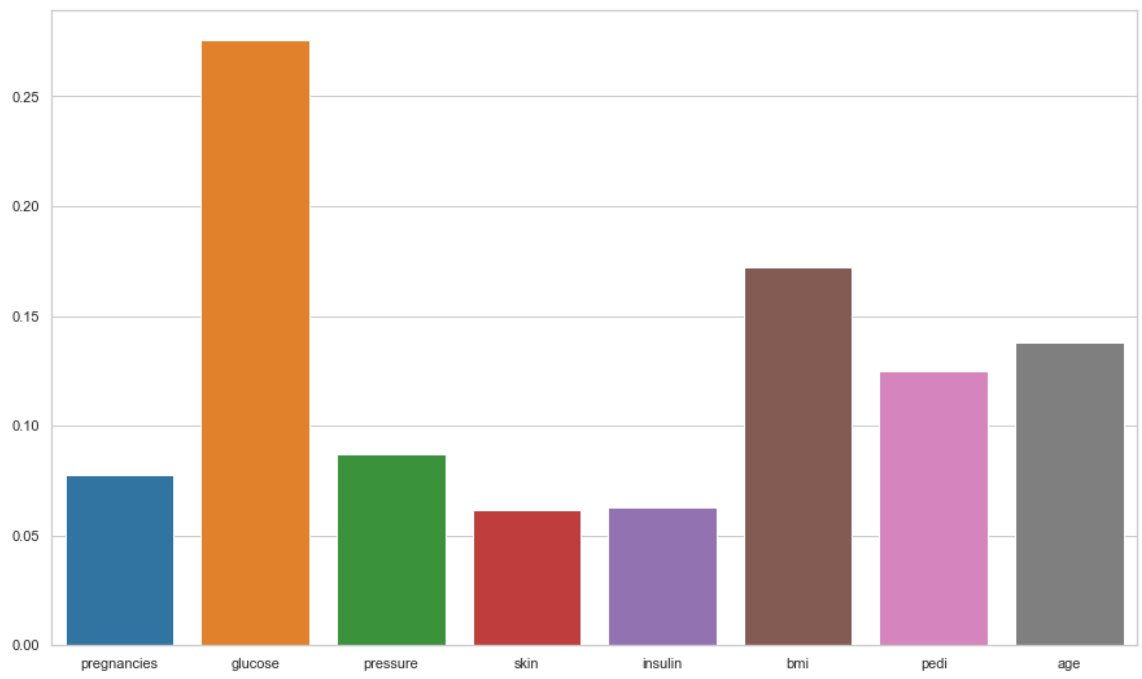
```
<Figure size 1080x648 with 0 Axes>
```

Out[11]:

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features=3,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=7, verbose
=0,
                       warm_start=False)

pregnancies     0.0778
glucose         0.2754
pressure        0.0873
skin            0.0617
insulin         0.0626
bmi             0.1721
pedi            0.1251
age             0.1379
```

Out[11]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fc81533388>
```



In [ ]:

Algorithm Comparison

In [12]:

```
# Now let's compare them all

plt.figure(figsize=(15,9))

sns.boxplot(data=resall, x="Type", y="Res")

sns.swarmplot(data=resall, x="Type", y="Res", color="royalblue")
```

Out[12]:

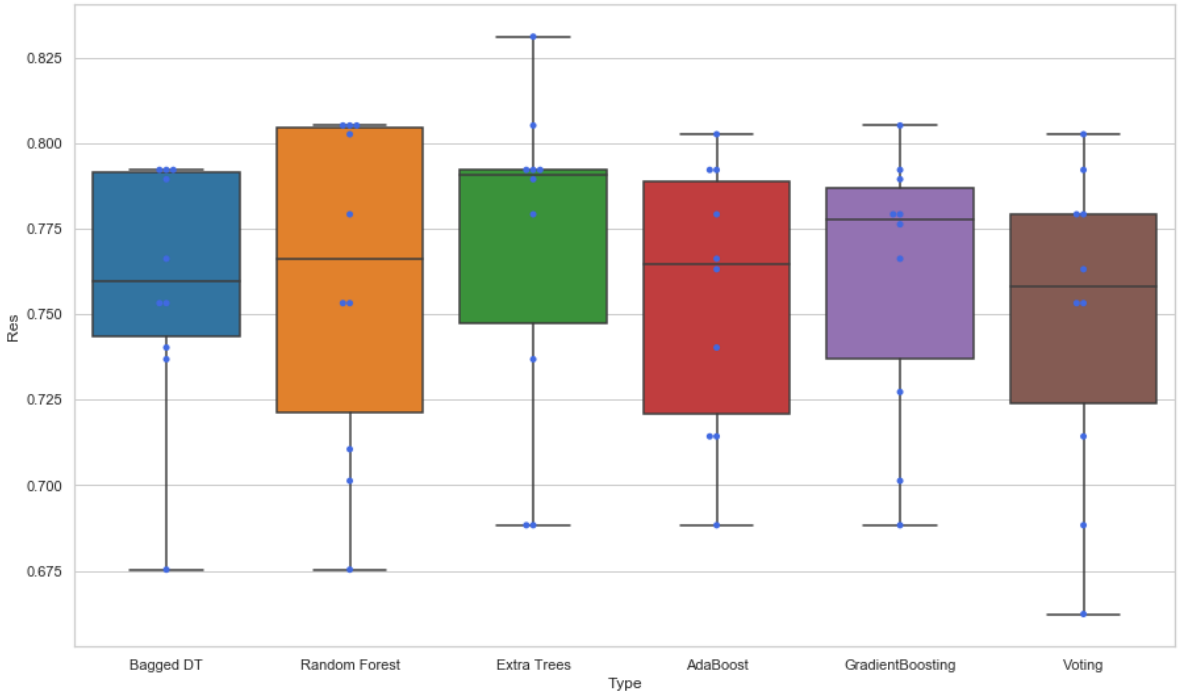
```
<Figure size 1080x648 with 0 Axes>
```

Out[12]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fc84602148>
```

Out[12]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fc84602148>
```



In [ ]:

Mission 1

a) Do the same with the Titanic dataset.

In [13]:

```
# Read the titanic data

titanic = pd.read_csv("titanic.csv")

# Make gender a dummy variable

titanic["Gender"] = titanic["Sex"].apply(lambda d: 1 if d == "female" else 0)

# Drop irrelevant variables, being sex and name

titanic.drop(["Name", "Sex"], axis = 1, inplace = True)

# Save y and then drop it from the dataframe

titanic_y = titanic["Survived"]
titanic.drop("Survived", axis = 1, inplace = True)

# Add y as the last

titanic["Survived"] = titanic_y

# Separate x and y

y = titanic["Survived"].values
x = titanic[["Pclass", "Age", "Siblings/Spouses Aboard", "Parents/Children Aboard", "Fare",
```

In [24]:

```
def ensembles(x,y):


    # Dataframes for results

    resall = pd.DataFrame()
    res_w1 = pd.DataFrame()


    # Determine parameters

    seed = 7
    kfold = KFold(n_splits=10, random_state=seed, shuffle = True)
    num_trees = 100


    # Bagged Decision Trees

    learner = DecisionTreeClassifier(random_state=seed)
    model = BaggingClassifier(base_estimator=learner, n_estimators=num_trees, random_state=

    results = cross_val_score(model, x, y, cv=kfold)
    res_w1["Res"] = results
    res_w1["Type"] = "Bagged DT"
    resall = pd.concat([resall,res_w1], ignore_index=True)


    # Random Forest

    num_features = 3
    model = RandomForestClassifier(n_estimators=num_trees, max_features=num_features, rando

    results = cross_val_score(model, x, y, cv=kfold)
    res_w1["Res"] = results
    res_w1["Type"] = "Random Forest"
    resall = pd.concat([resall,res_w1], ignore_index=True)


    # Redefining number of trees and features

    num_trees = 300
    num_features = 5


    # Extra Trees

    model = ExtraTreesClassifier(n_estimators=num_trees, max_features=num_features, random_

    results = cross_val_score(model, x, y, cv=kfold)
    res_w1["Res"] = results
    res_w1["Type"] = "Extra Trees"
    resall = pd.concat([resall,res_w1], ignore_index=True)


    # ADA Boost

    num_trees = 30

    model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed)
```

```
results = cross_val_score(model, x, y, cv=kfold)
res_w1["Res"] = results
res_w1["Type"] = "AdaBoost"
resall = pd.concat([resall,res_w1], ignore_index=True)


# Stochastic Gradient Boosting

num_trees = 30

model = GradientBoostingClassifier(n_estimators=num_trees, random_state=seed)

results = cross_val_score(model, x, y, cv=kfold)
res_w1["Res"] = results
res_w1["Type"] = "GradientBoosting"
resall = pd.concat([resall,res_w1], ignore_index=True)


# Voting Ensemble

estimators = []
model1 = LogisticRegression(solver="liblinear")
estimators.append(("logistic", model1))

model2 = DecisionTreeClassifier(random_state=seed)
estimators.append(("cart", model2))

model3 = SVC(gamma="auto")
estimators.append(("svm", model3))

num_trees = 100
num_features = 3

model4 = RandomForestClassifier(n_estimators=num_trees, max_features=num_features, rand
estimators.append(("rfc", model4))

model = VotingClassifier(estimators)

results = cross_val_score(model, x, y, cv=kfold)
res_w1["Res"] = results
res_w1["Type"] = "Voting"
resall = pd.concat([resall,res_w1], ignore_index=True)


# Barplot

plt.figure(figsize=(15,9))
seed = 7

num_trees = 100
num_features = 3
model = RandomForestClassifier(n_estimators=num_trees, max_features=num_features, rando
model.fit(x,y)

for name, importance in zip(titanic.columns, model.feature_importances_):
    print(f'{name:15s}  {importance:.4f}')

sns.barplot(x=titanic.columns[:-1], y = model.feature_importances_)
```

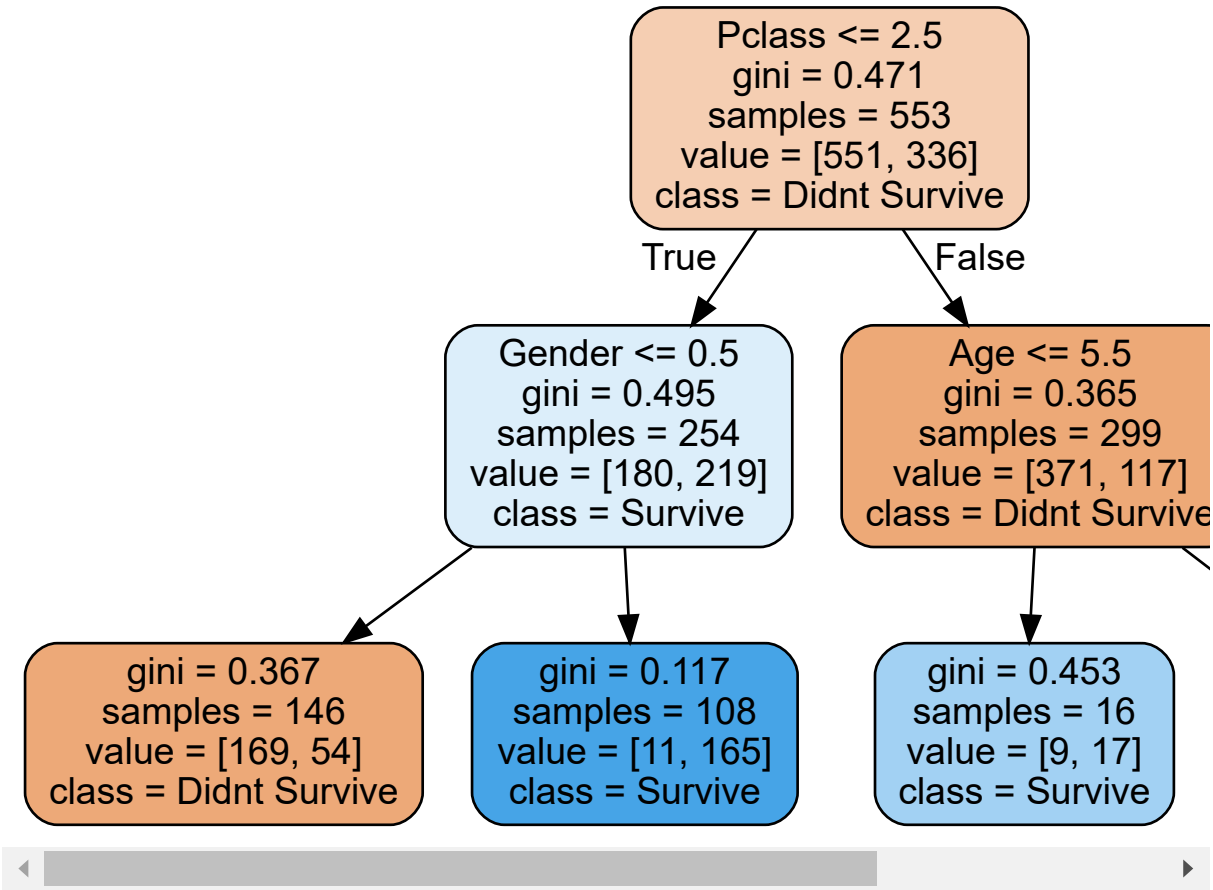
```
# Algorithm Comparison

plt.figure(figsize=(15,9))
sns.boxplot(data=resall, x="Type", y="Res")
sns.swarmplot(data=resall, x="Type", y="Res", color="royalblue")


# Plotting the tree

seed=7

num_trees=100
num_features=3

# Max_depth set at 2 for visualization purposes

model=RandomForestClassifier(n_estimators=num_trees, max_depth = 2, max_features=num_fe
model.fit(x,y)

estimator = model.estimators_[5]

graph=Source(tree.export_graphviz(estimator,
        out_file=None,
        feature_names = ["Pclass", "Age", "Siblings/Spouses Aboard", "Parents/Children
        class_names=["Didnt Survive", "Survive"],
        filled=True,
        rounded=True))
return graph
```

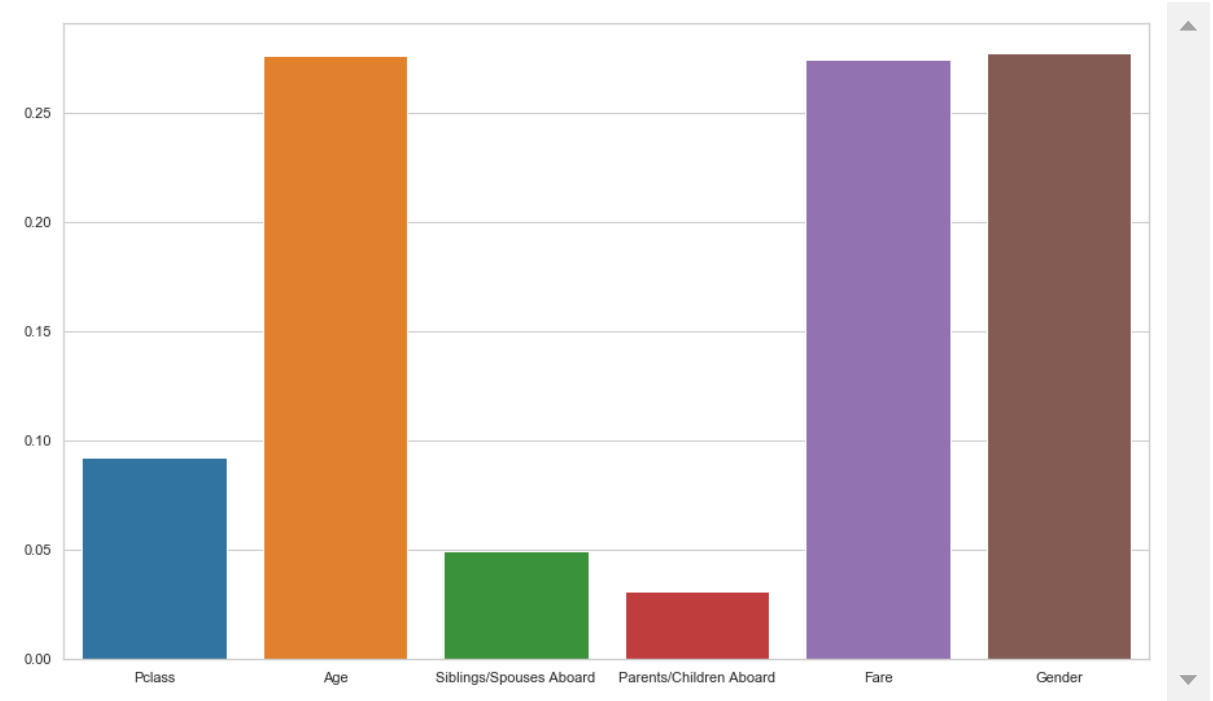
In [25]:

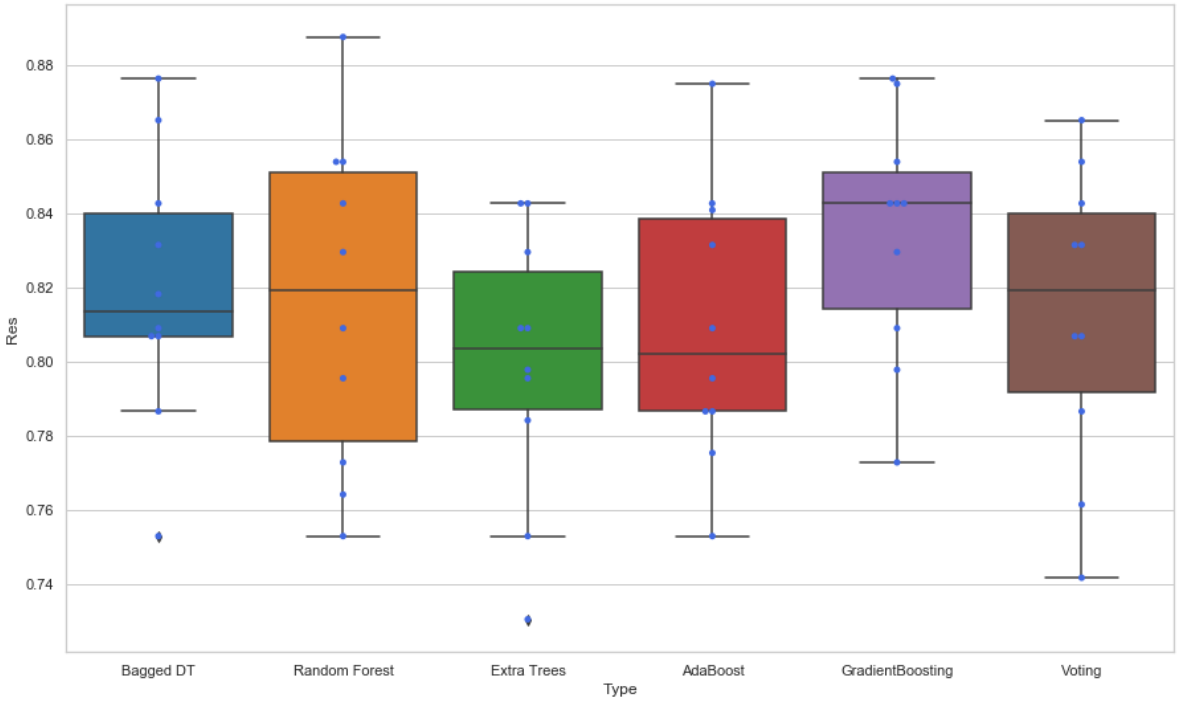
```
ensembles(x,y)
```

```
Pclass              0.0925
Age                 0.2759
Siblings/Spouses Aboard  0.0494
Parents/Children Aboard  0.0308
Fare                0.2744
Gender              0.2769
```

Out[25]:






0.88
0.86
0.84
0.82
0.80
0.78
0.76
0.74
Res
Bagged DT
Random Forest
Extra Trees
AdaBoost
GradientBoosting
Voting
Type

In [ ]: