Prediction of turtle sex with Machine Learning

What machine learning algorithms do is - in principle - very simple: they are fed with data, that they use to define a *decision surface*, mainly used to cluster or tag new/unknown data. We're going to use turtle shells metrics to try and predict turtles sex.

Supervised algorithms make use of labelled data, **unsupervised** ones try to cluster the data by the means of mathematics and distance parameters.

To start scratching the surface, I recommend the following Udemy's course: [https://classroom.udacity.com/courses/ud120 (https://classroom.udacity.com/courses/ud120)](https://classroom.udacity.com/courses/ud120)

What we will learn

With Python' we'll get to making predictions on actual data, by leveraging Principal Component Analysis (PCA) and Machine Learning (ML) algorithms.

This is a very tiny dataset, but comes from real data, it's handy to work with and it is as good as any as a starter for learning *sklearn* syntax.

Explanation of the inner clockwork of each ML algorithm as well as algorithm comparison is beyond the scope of this notebook.

Limitations

i) Please note that PCA is not *needed*, and there are better and optimized ways to integrate it in your workflow (for example, [pipeline (http://scikit-learn.org/stable/modules/pipeline.html)](http://scikit-learn.org/stable/modules/pipeline.html), but this is out of the scope of this notebook. **ii)** Also outside the scope of this notebook is automating the choice of the algorithm and/or of its parameters, achievable through [GridSearchCV (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html). **iii)** Last, we will not be discussing strategies on how to split and make use of all data for training and testing our algorithms.

In [1]:

```
%matplotlib inline
import pandas as pd
import numpy as np

# plotting
from matplotlib import pyplot as plt
import seaborn as sns

sep = "\t"
```

Data preprocessing

Dataset splitting into training and testing

We need to **split our dataset in two**: we'll be using 75% of our entries to **train** the algorithm, that will subsequently try and predict the labels of our remaining **test** data.

It is good practice, when transforming the input data (and in this example we'll be both [rescaling and applying PCA (https://github.com/Stemanz/python_tutorials/tree/master/PCA)](https://github.com/Stemanz/python_tutorials/tree/master/PCA), to fit the transformation algorithm on the train set, then applying the same transformation obtained from that fit to both train and test sets. Applying data transformation to all points beforehand can potentially lead to altered results.

In [2]:

```
# load data
df = pd.read_csv("turtles.csv", sep=sep)

# for easy handling, we separe data (numerical features) and labels.
labels = list(df["Sex"])

# divide data into train and test. Lengthy line -.-'
from sklearn.model_selection import train_test_split

data_train, data_test, labels_train, labels_test = train_test_split(
                                                       df, labels, test_size=.25
,
                                                       random_state=24
                                                   )
```

`train_test_split` spits back the same type of container it was fed with, so we get back a `DataFrame` for the numerical data, randomly chosen:

In [3]:

```
data_train.head()
```

Out[3]:

	ID	Sex	length	width	height
26	T3	M	96	80	35
28	T5	M	102	85	38
10	T35	F	134	100	48
33	T10	M	112	89	40
47	T24	M	135	106	47

The labels are still within a `list` .

Data transformation

Now we can proceed and try to prepare the data for the subsequent analysis, by rescaling and reducing the number of dimensions. `sklearn` does not processes `DataFrames` , so we'll get a numpy representation of it by `DataFrame.values` .

In [4]:

```
# After dividing original data into train and test sets, we transform data
# by fitting over the train set, and transforming both trand AND test sets
# with that fit

# scale data
from sklearn.preprocessing import StandardScaler

features = ["length", "width", "height"]
data_train = data_train.loc[:, features].values #numpy-style
data_test = data_test.loc[:, features].values
import warnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore") # int64 to float64 warning suppression
    scaler = StandardScaler().fit(data_train)
    data_train_scaled = scaler.transform(data_train)
    data_test_scaled = scaler.transform(data_test)


from sklearn.decomposition import PCA

# PCA dimensionality reduction: 3 to 2
pca = PCA(n_components=2, whiten=True)
PC = pca.fit(data_train_scaled)
data_train_PC = pca.transform(data_train_scaled)
data_test_PC = pca.transform(data_test_scaled)

# we're ready!
```

Supervised ML algorithms

Naïve Bayes

For each algorithm, if you are interested in the inner workings or the parameters, watch some youtube videos and read *sklearn* documentation. In here I'll cover **how they are deployed**.

In [5]:

```
from sklearn.naive_bayes import GaussianNB # NB stands for Naïve Bayes

# training
clf = GaussianNB()

# some downstream functions *require* labels to be integeres in np.array
# we will "retranslate" numerical IDs to sexes when plotting
labels_train = np.array([0 if x=="M" else 1 for x in labels_train])
labels_test = np.array([0 if x=="M" else 1 for x in labels_test])

clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)
```

Conveniently, different ML algorithms do have the same syntax. The class is instantiated by calling it (with or withour parameters), then it is fitted with `.fit()` . Predictions are made via `.predict()` .

If the array is conveniently shaped, it is possible to batch-predict stuff. Also individual points can be predicted, for instance:

In [6]:

```
clf.predict([[ 0.02736815,  2.47830153]]) # note the double square brackets
```

Out[6]:

```
array([0])
```

Checking the accuracy of predictions

It is crucial to determine the accuracy of the model. *sklearn* provides a tool that is cross-algorithm.

In [7]:

```
from sklearn.metrics import accuracy_score
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
91.67%
```

Without any help and by just using real world measures and labels, the trained algorithm was able to correctly predict the sex of 91% unknown samples. Not a bad thing!

Plotting the data

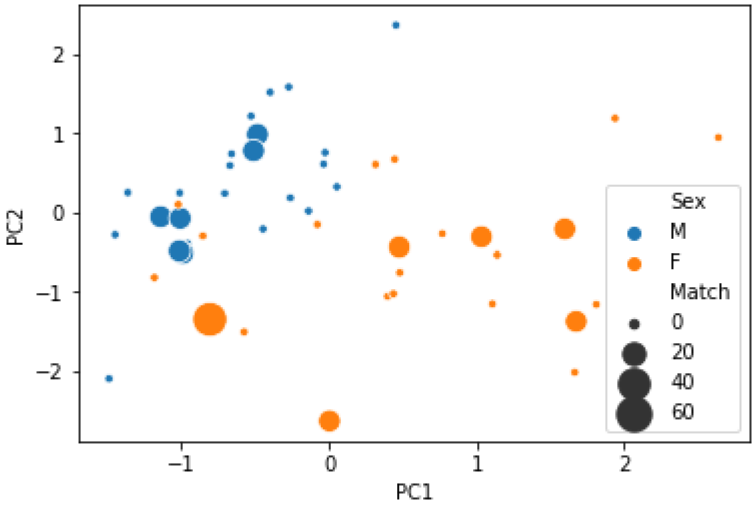
Let's visualize it (also, making a function to repack data back to a DataFrame, which I'll be reusing throughout the notebook):

In [8]:

```
# Please note that this function pulls stuff from the global namespace
# as it is only intended to be used in this notebook, so there's no need
# to raise any eyebrows.
def repack_results(train_size=1, ok_size=20, ko_size=50):
    """
    I'm converting back the arrays to a dataframe.
    Also, I am giving a large numerical value where predictions DO NOT match input data
    """
    traindata = pd.DataFrame(data_train_PC, columns=["PC1", "PC2"])
    traindata["Sex"] = ["M" if x==0 else "F" for x in labels_train]
    traindata["Match"] = [train_size for _ in range(traindata.shape[0])]

    results = pd.DataFrame(data_test_PC, columns=["PC1", "PC2"])
    results["Sex"] = ["M" if x==0 else "F" for x in labels_test]
    results["Match"] = [ok_size if x == y else ko_size for x,y in zip(labels_test, labels_pred)]

    return pd.concat([results, traindata]).sort_values(by="Sex", ascending=False)

res = repack_results()
```

In [9]:

```
ax = sns.scatterplot(data=res, x="PC1", y="PC2", hue="Sex", size="Match", sizes=(20,300))
```



This plot shows the training dataset as tiny spots. Larger spots are the ones that have been used as test and correctly predicted. The biggest ones are the ones whose prediction does not match the labels. As the dataset is split randomly each time, results and predictions may vary in your analysis (in mine, I've set the random number generator seed, so each time the random choices will be the same).

In [10]:

```
# let's see where test and predicted labels differ

print("Point", "Truth", "Pred", "Match", sep=sep)
for i, elems in enumerate(zip(labels_test, labels_pred)):
    t, p = elems
    if t != p:
        error = "MISPREDICTION!"
    else:
        error = ""

    print(i, t, p, error, sep=sep)
```

```
Point   Truth   Pred    Match
0       1       1
1       1       1
2       0       0
3       0       0
4       1       1
5       1       0       MISPREDICTION!
6       1       1
7       0       0
8       0       0
9       0       0
10      1       1
11      0       0
```

It looks like the algorithm misdiagnosed that point in the middle of the plot: anyway, it would have been tough even for a human!

Plotting the decision surface

Last, we can actually visualize the decision boundary of the machine learning algorithm. There's no need to get mad and implement that ourselves, as *mlxtend* does the job for us. It needs the original data and our trained classifier, and its use is super easy.

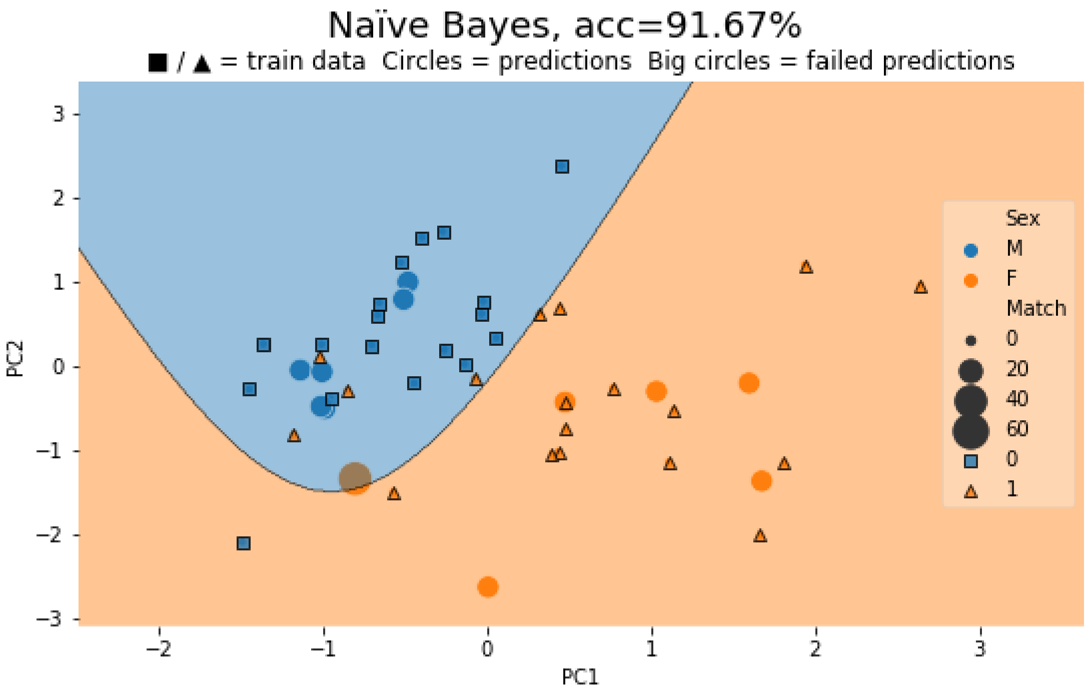
You can install it via the command line:

```
pip install mlxtend
```

Here's some lines that I'm packing into a function for later use.

In [11]:

```
from mlxtend.plotting import plot_decision_regions

# see comments for the previous function! :)
suptitle = f"Naïve Bayes, acc={round(acc*100, 2)}%"
def plot_results():
    fig = plt.figure(figsize=(9,5))
    fig.suptitle(suptitle, size=18)
    ax = sns.scatterplot(data=res, x="PC1", y="PC2", hue="Sex", size="Match", si
zes=(20,300))
    ax.set_title("▪ / ▲ = train data  Circles = predictions  Big circles = faile
d predictions")

    plot_decision_regions(data_train_PC, labels_train, clf=clf, legend=5)

plot_results()
```



That is the reason of the misprediction.

SVM (Support Vector Machines)

We now go over the algorithms quicker, as the same general scheme is followed.

Note that SVM expects that the data is *linearly separable*. When it is not, tricks can be employed to make it linearly separable (i.e. squaring variables, taking the modulus of variables, and so on.). These are referred to as *kernel tricks*. Try some different to find the one that suits your data the most!

In [12]:

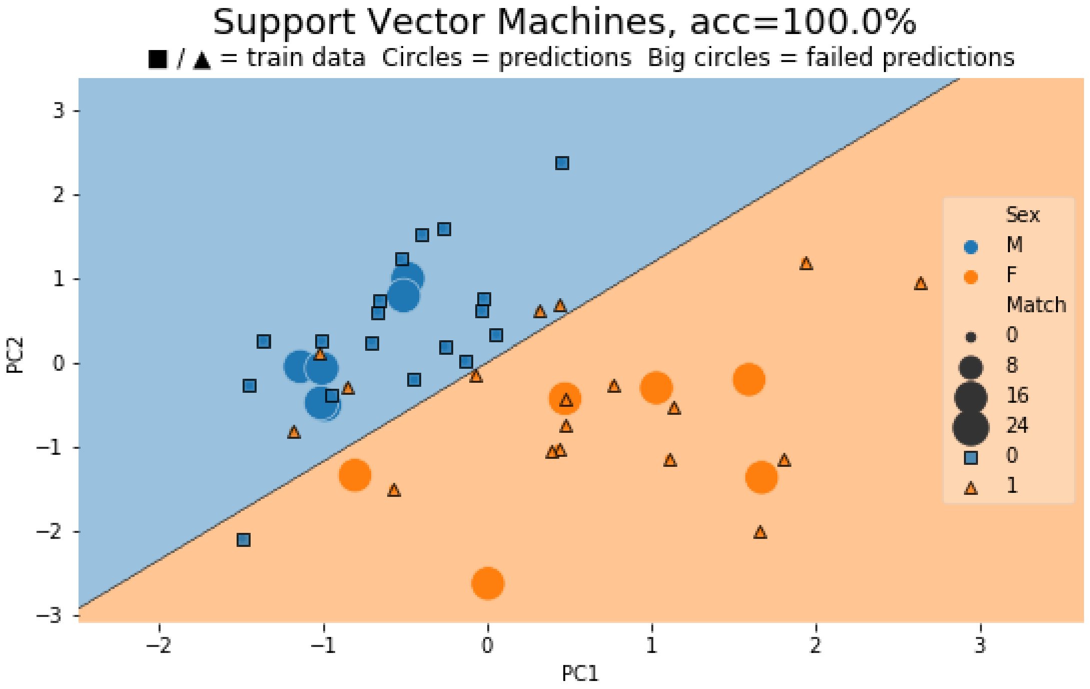
```
from sklearn.svm import SVC

# training
clf = SVC(kernel="linear")
clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
100.0%
```

Let's see how the decision boundary looks like.

In [13]:

```
res = repack_results()

suptitle = f"Support Vector Machines, acc={round(acc*100, 2)}%"
plot_results()
```



We're now trying with a new kernel. From the doc:

> Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable.

Conveniently, *callable* is a custom function. We're going to stick with builtin ones for now ;)

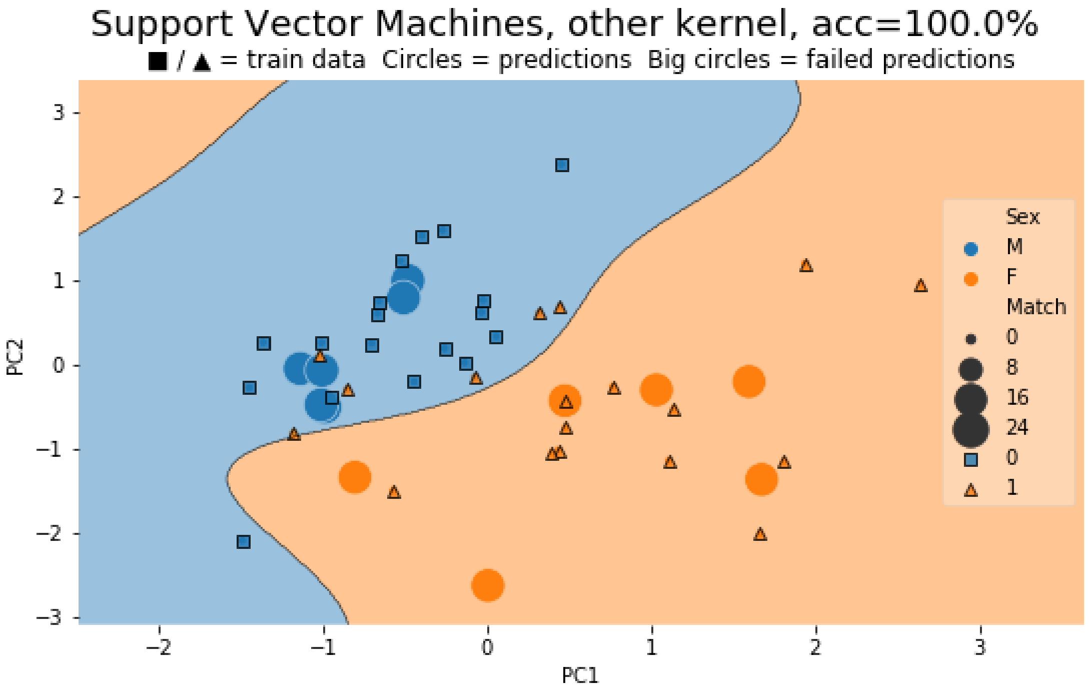
In [14]:

```
# SVC training with another kernel

# training
clf = SVC(kernel="rbf", gamma="scale")
clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
100.0%
```

In [15]:

```
res = repack_results()

suptitle = f"Support Vector Machines, other kernel, acc={round(acc*100, 2)}%"
plot_results()
```



As you can see, kernel tricks do play a **huge** role in defining SVM behavior!

Other parameters that should be taken into account are *C* and *gamma*. Roughly, *C* controls the tradeoff between having a smooth decision boundary and classifying the points correctly. Whereas, *gamma* defines how far each single training point weighs into defining the decision boundary.

As always, get your hands dirty into changing parameters and see how they shape the results and the decision boundary.

Please note that classifying correctly all the points, at the cost of generating a very wiggly decision boundary, is ***not*** what we aim at. Tweaking the parameters too much can result in **overfitting**, which should be avoided.

Small recap of important SVM parameters

kernel

C

gamma

Decision Tree

Decision Tree is a really interesting algorithm that "grows" trees based on decisions, which tries to obtain, iteration through iteration, subsets as *pure* as possible. Each tree has *branches* and *leaves*, whose number can be tuned its behavior.

In [16]:

```
from sklearn import tree

# training
clf = tree.DecisionTreeClassifier()
clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
66.67%
```

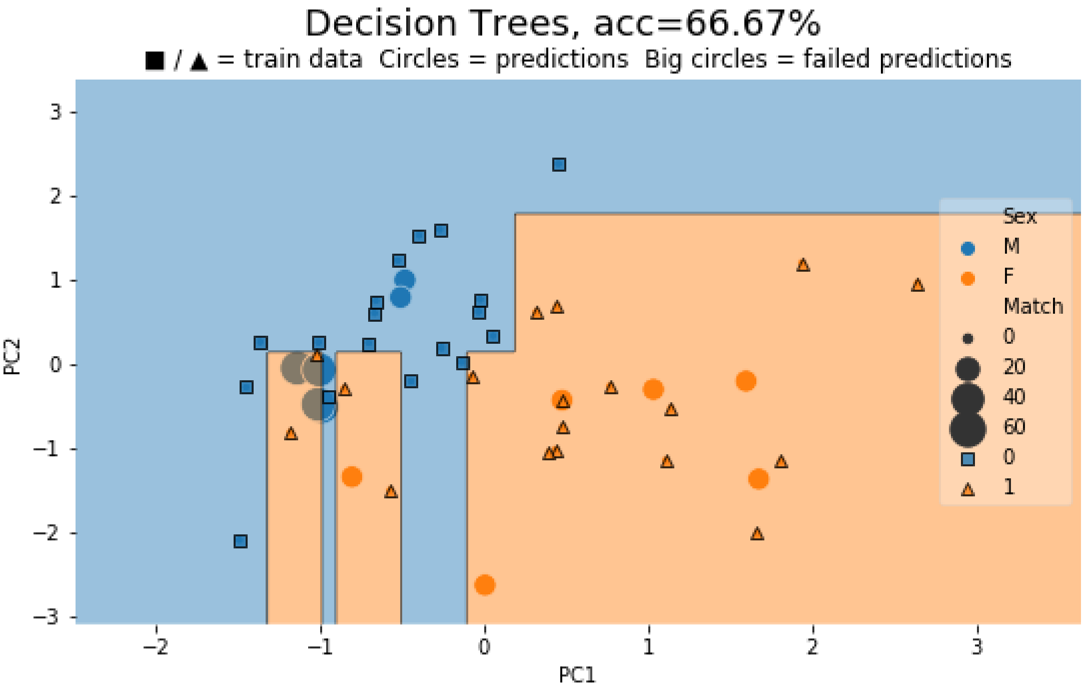
Well, well. Depending on the run, I usually get around 70% accuracy. Why is this changing? Because decision trees are grown randomly, and the final predictor can change even if trained with the same data!

The classifier has the *random_state* parameter, that can be used to seed the random number generator and obtain reproducible results (not done by me in this example).

Let's plot:

In [17]:

```
res = repack_results()

suptitle = f"Decision Trees, acc={round(acc*100, 2)}%"
plot_results()
```



In this case, 2 or 3 samples have been incorrectly predicted. As we discussed briefly, the algorithm can be tuned in different ways:

- Adjusting the number of leaves. Each "leaf" is a sample at the end of all iterations. It defaults to 1, but it can be increased via *min_samples_leaf*
- Choosing when stop splitting. Many branches are created as long as there are samples that remain to be split. The parameter *min_samples_split* sets the number of samples below which the algorithm would refrain from splitting data further. It sorta complements the previous parameter.
- Tweaking the splitting algorithm, the criteria used, the allowed number of branches, iterations, and so on.

Please do refer to the excellent *sklearn* doc for both algorithm explanation and the many other parameters.

Random Forest

This algorithm leverages on the latter. It is an ***ensemble method***, as it uses many times another classifier (in this case, Decision Tree) on a subset of data, then chooses the answer for predicting new data that's more likely among the host of miniclassifiers. Aptly, it's been called Random Forest.

In [18]:

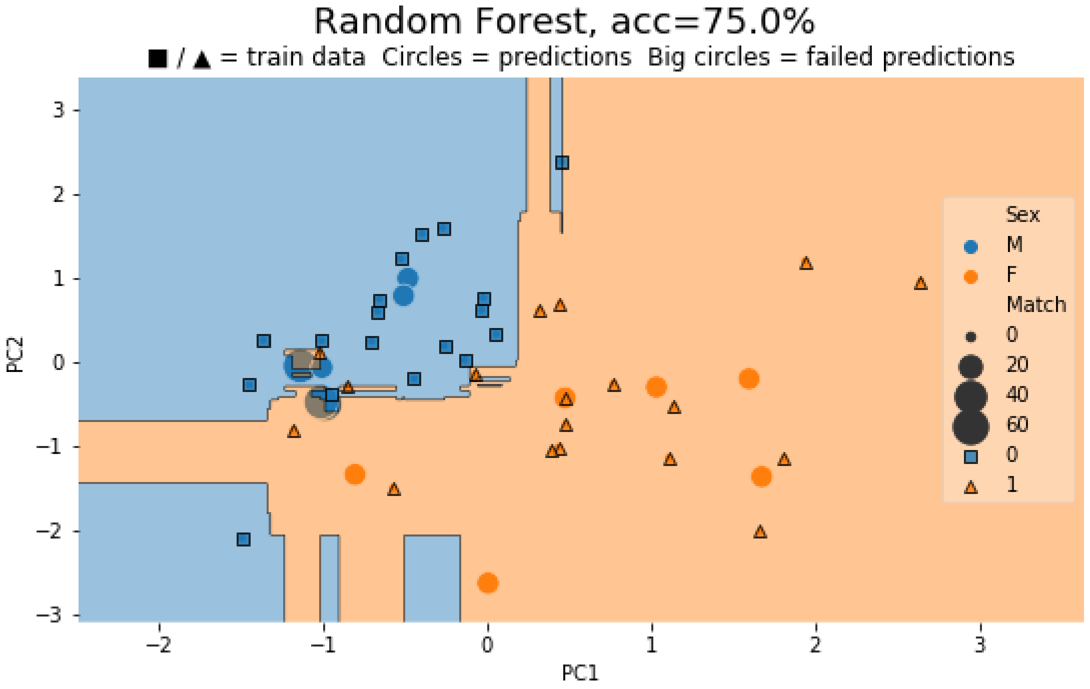
```
from sklearn.ensemble import RandomForestClassifier

# training
clf = RandomForestClassifier(n_estimators=100)
clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
75.0%
```

n_estimators clearly sets the size of the forest ;)

In [19]:

```
res = repack_results()

suptitle = f"Random Forest, acc={round(acc*100, 2)}%"
plot_results()
```



Many parameters have a 1-to-1 correspondence with Decision Tree algorithm. Let's see if we can tweak it a bit:

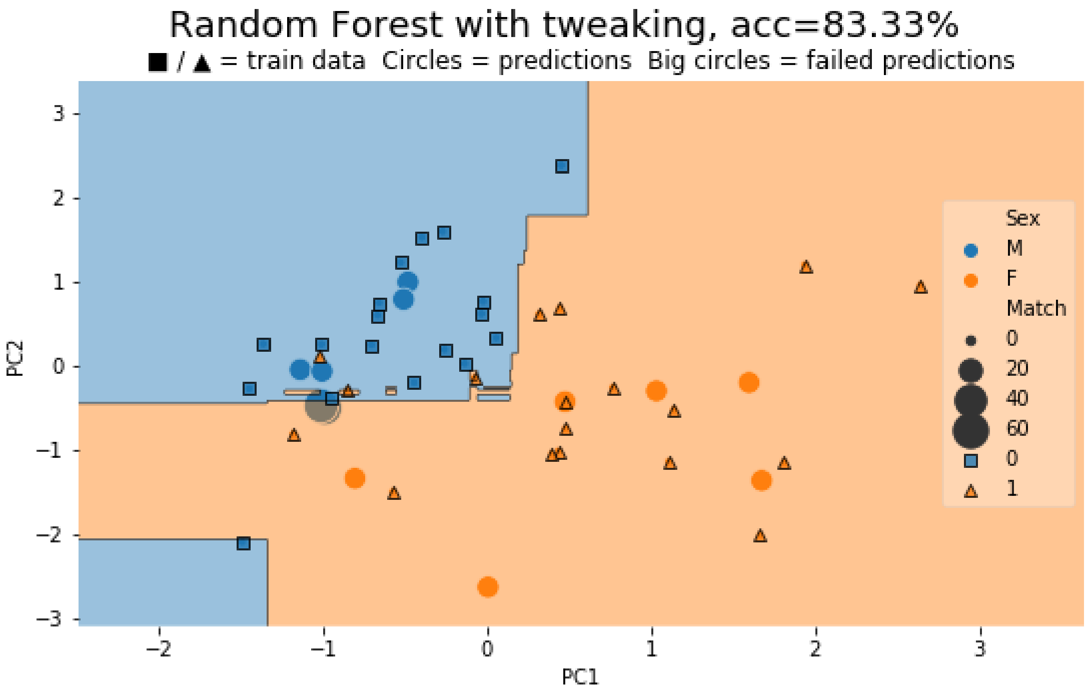
In [20]:

```
params = {
    'min_impurity_decrease': 0.005,
    'min_samples_leaf': 1,
    'min_samples_split': 6,
    'n_estimators': 50
}

clf = RandomForestClassifier(**params)

clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
83.33%
```

In [21]:

```
res = repack_results()

suptitle = f"Random Forest with tweaking, acc={round(acc*100, 2)}%"
plot_results()
```



This might be better in terms of precision, but the decision surface looks like it's not optimal. Parameter tweaking can lead to greater precision but also to lose real predictive power against new, never-seen-before data points. Always beware of **overfitting**!

Adaboost

Adaboost, short for *adaptive boosting*, is another ***ensemble method*** as it can use other machine learning algorithms of choice to make final decisions about new data.

The algorithm of choice defaults to Decision Tree, here's how to swith it towards another classifier (Support Vector Machine classifier, for example):

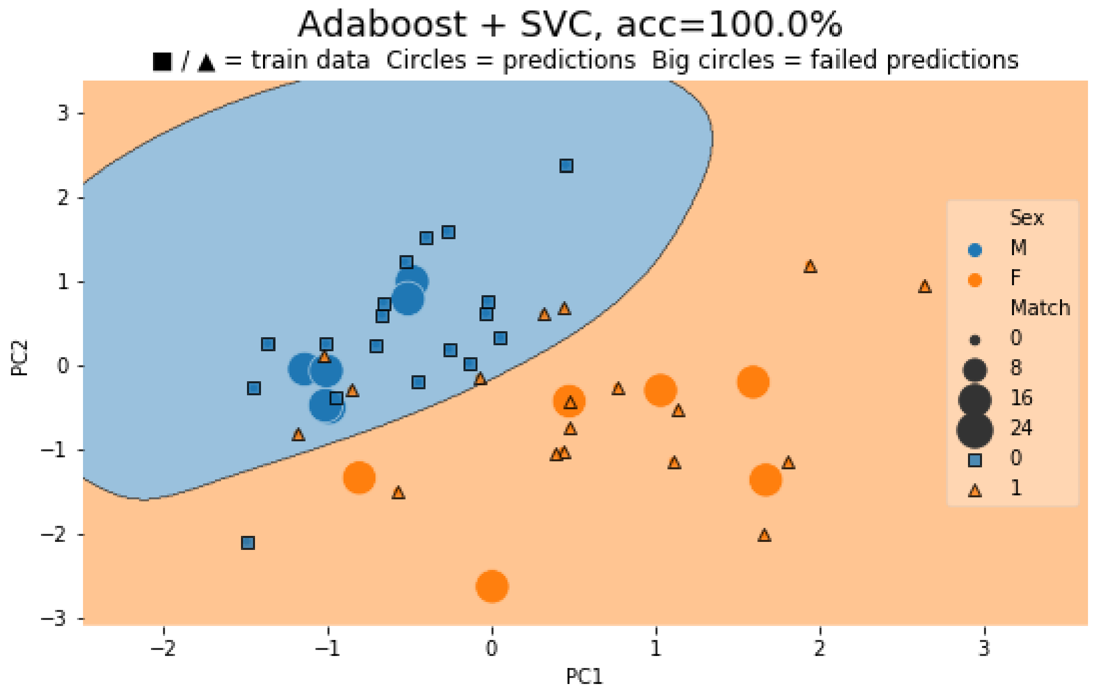
In [28]:

```
from sklearn.ensemble import AdaBoostClassifier

# training
clf = AdaBoostClassifier(
    base_estimator=SVC(gamma="scale"),
    n_estimators=200,
    algorithm="SAMME"
)
clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
100.0%
```

In [27]:

```
res = repack_results()

suptitle = f"Adaboost + SVC, acc={round(acc*100, 2)}%"
plot_results()
```



KNN

KNN's not a secret set, it stands for K-nearest neighbor. While described as "the most simple" ML algorithm, that is not apparent in its Wikipedia description. Let's get to the syntax:

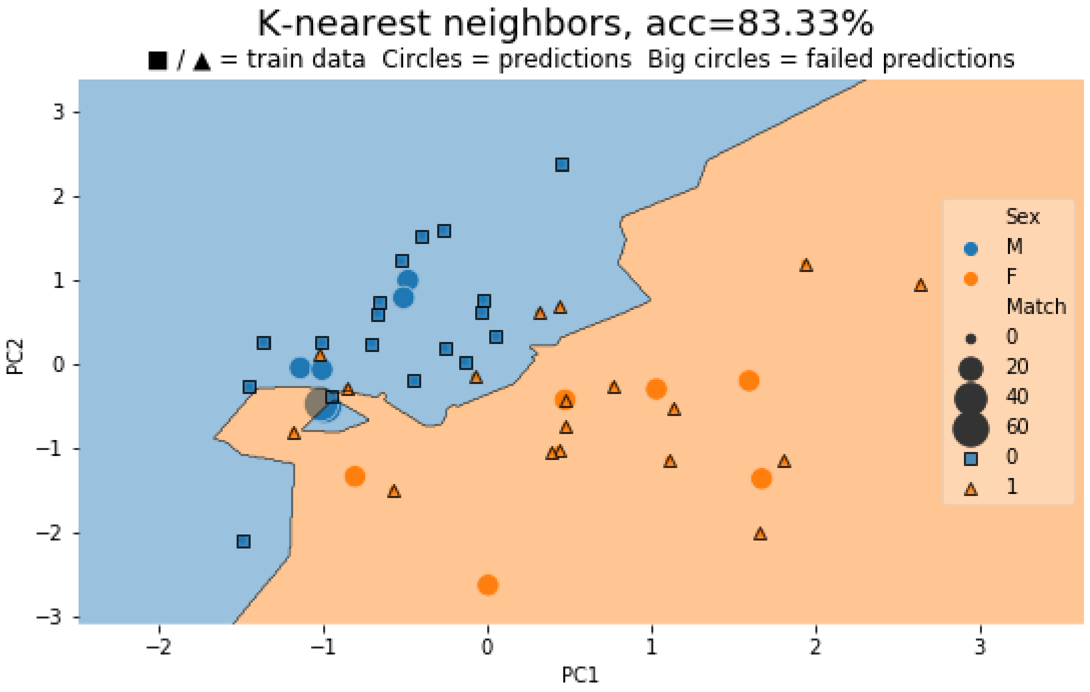
In [24]:

```
from sklearn.neighbors import KNeighborsClassifier as KNN

# training
clf = KNN()
clf.fit(data_train_PC, labels_train)

# predicting
labels_pred = clf.predict(data_test_PC)

# testing the accuracy of predictions
acc = accuracy_score(labels_pred, labels_test)
print(round(acc*100, 2), "%", sep="")
```

```
83.33%
```

In [25]:

```
res = repack_results()

suptitle = f"K-nearest neighbors, acc={round(acc*100, 2)}%"
plot_results()
```



In [ ]:

This ends our little syntax tour towards analyzing our data with *sklearn* and supervised classification algorithms. I would recommend you to read about the *logic* each classifier uses, even if it is not necessary to understand all the math (as it is not necessary to being able coding in C to use any operating system).

Thank you for bearing with me all along!

Final remarks:

- RTFM! always. (read the freakin' manual - aka the doc)
- Actively play with your data. Change parameters. Break things and see what happens.
- Do not take all these examples (on such a limited dataset) as an algorithm *competition*. There's no such thing as best algorithm, every situation needs a different approach.
- Every algorithm has a parameter space. *Finding the best one is the art at the bottom of machine learning*.
- There'll be people tampering with your data, but they **do not care** and **they make mistakes they don't care about**. Be in control of your data!