Public health awareness campaign analysis

# Introduction :

The objective of this project is to provide an in-depth analysis of the design and innovation strategies for developing a public health awareness campaign using data analytics with cognos tool. A public health awareness campaign is a powerful tool for promoting well-being within communities and this project aims to utilize innovative approaches to enhance prediction accuracy and reliability.

# Problem Statement :

The tech industry has been known for its fast-paced and demanding work environment, which can often lead to increased stress and mental health challenges among its employees. It is essential to understand the extent of these issues, identify contributing factors, and explore potential solutions to improve the mental well-being of tech professionals. This survey aims to investigate the current state of mental health in the tech industry the factors affecting it, and the attitudes towards seeking help and support among tech workers.

**Importance of loading and processing dataset:**

Loading and preprocessing the dataset is an important first step inbuilding any machine learning model. However, it is especiallyimportant for house price prediction models, as house price datasets areoften complex and noisy. By loading and preprocessing the dataset, we can ensure that themachine learning algorithm is able to learn from the data effectively andaccurately.

**Preprocessing the dataset:**

Data preprocessing is the process of cleaning, transforming, andintegrating data in order to make it ready for analysis.

This may involve removing errors and inconsistencies, handlingmissing values, transforming the data into a consistent format, andscaling the data to a suitable range.

## The proccess is the following:

* [Library and data loading](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Library_and_data_loading)
* [Data cleaning](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Data_cleaning)
* [Encoding data](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Encoding_data)
* [Covariance Matrix. Variability comparison between categories of variables](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Covariance_Matrix)
* [Some charts to see data relationship](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Some_charts_to_see_data_relationship)
* [Scaling and fitting](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Scaling_and_fitting)
* [Tuning](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Tuning)
* [Random Forests](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Random_Forests)
* [Bagging](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Bagging)
* [Boosting](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Boosting)
* [Stacking](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Stacking)
* [Success method plot](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Success_method_plot)
* [Creating predictions on test set](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Creating_predictions_on_test_set)
* [Conclusions](https://www.kaggle.com/code/sabbirshibli/machine-learning-for-mental-health#Conclusions)

## ****Library and data loading****

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

from scipy.stats import randint

from sklearn.model\_selection import train\_test\_split

from sklearn import preprocessing

from sklearn.datasets import make\_classification

from sklearn.preprocessing import binarize, LabelEncoder, MinMaxScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier

from sklearn import metrics

from sklearn.metrics import accuracy\_score, mean\_squared\_error, precision\_recall\_curve

from sklearn.model\_selection import cross\_val\_score

from sklearn.neural\_network import MLPClassifier

from sklearn.grid\_search import RandomizedSearchCV

from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.naive\_bayes import GaussianNB

from mlxtend.classifier import StackingClassifier

from subprocess import check\_output

train\_df = pd.read\_csv('../input/survey.csv')

print(train\_df.shape)

print(train\_df.describe())

print(train\_df.info())

## ****Data cleaning****

total = train\_df.isnull().sum().sort\_values(ascending=False)

percent = (train\_df.isnull().sum()/train\_df.isnull().count()).sort\_values(ascending=False)

missing\_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])

missing\_data.head(20)

print(missing\_data)

train\_df = train\_df.drop(['comments'], axis= 1)

train\_df = train\_df.drop(['state'], axis= 1)

train\_df = train\_df.drop(['Timestamp'], axis= 1)

train\_df.isnull().sum().max() *#just checking that there's no missing data missing...*

train\_df.head(5)

**Cleaning NaN**

defaultInt = 0

defaultString = 'NaN'

defaultFloat = 0.0

intFeatures = ['Age']

stringFeatures = ['Gender', 'Country', 'self\_employed', 'family\_history', 'treatment', 'work\_interfere',

'no\_employees', 'remote\_work', 'tech\_company', 'anonymity', 'leave', 'mental\_health\_consequence',

'phys\_health\_consequence', 'coworkers', 'supervisor', 'mental\_health\_interview', 'phys\_health\_interview',

'mental\_vs\_physical', 'obs\_consequence', 'benefits', 'care\_options', 'wellness\_program',

'seek\_help']

floatFeatures = []

for feature **in** train\_df:

if feature **in** intFeatures:

train\_df[feature] = train\_df[feature].fillna(defaultInt)

elif feature **in** stringFeatures:

train\_df[feature] = train\_df[feature].fillna(defaultString)

elif feature **in** floatFeatures:

train\_df[feature] = train\_df[feature].fillna(defaultFloat)

else:

print('Error: Feature **%s** not recognized.' % feature)

train\_df.head(5)

gender = train\_df['Gender'].str.lower()

gender = train\_df['Gender'].unique()

male\_str = ["male", "m", "male-ish", "maile", "mal", "male (cis)", "make", "male ", "man","msle", "mail", "malr","cis man", "Cis Male", "cis male"]

trans\_str = ["trans-female", "something kinda male?", "queer/she/they", "non-binary","nah", "all", "enby", "fluid", "genderqueer", "androgyne", "agender", "male leaning androgynous", "guy (-ish) ^\_^", "trans woman", "neuter", "female (trans)", "queer", "ostensibly male, unsure what that really means"]

female\_str = ["cis female", "f", "female", "woman", "femake", "female ","cis-female/femme", "female (cis)", "femail"]

for (row, col) **in** train\_df.iterrows():

if str.lower(col.Gender) **in** male\_str:

train\_df['Gender'].replace(to\_replace=col.Gender, value='male', inplace=True)

if str.lower(col.Gender) **in** female\_str:

train\_df['Gender'].replace(to\_replace=col.Gender, value='female', inplace=True)

if str.lower(col.Gender) **in** trans\_str:

train\_df['Gender'].replace(to\_replace=col.Gender, value='trans', inplace=True)

stk\_list = ['A little about you', 'p']

train\_df = train\_df[~train\_df['Gender'].isin(stk\_list)]

print(train\_df['Gender'].unique())

train\_df['Age'].fillna(train\_df['Age'].median(), inplace = True)

s = pd.Series(train\_df['Age'])

s[s<18] = train\_df['Age'].median()

train\_df['Age'] = s

s = pd.Series(train\_df['Age'])

s[s>120] = train\_df['Age'].median()

train\_df['Age'] = s

train\_df['age\_range'] = pd.cut(train\_df['Age'], [0,20,30,65,100], labels=["0-20", "21-30", "31-65", "66-100"], include\_lowest=True)

train\_df['self\_employed'] = train\_df['self\_employed'].replace([defaultString], 'No')

print(train\_df['self\_employed'].unique())

train\_df['work\_interfere'] = train\_df['work\_interfere'].replace([defaultString], 'Don**\'**t know' )

print(train\_df['work\_interfere'].unique())

## ****Encoding data****

labelDict = {}

for feature **in** train\_df:

le = preprocessing.LabelEncoder()

le.fit(train\_df[feature])

le\_name\_mapping = dict(zip(le.classes\_, le.transform(le.classes\_)))

train\_df[feature] = le.transform(train\_df[feature])

labelKey = 'label\_' + feature

labelValue = [\*le\_name\_mapping]

labelDict[labelKey] =labelValue

for key, value **in** labelDict.items():

print(key, value)

train\_df = train\_df.drop(['Country'], axis= 1)

train\_df.head()

### Testing there aren't any missing data

total = train\_df.isnull().sum().sort\_values(ascending=False)

percent = (train\_df.isnull().sum()/train\_df.isnull().count()).sort\_values(ascending=False)

missing\_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])

missing\_data.head(20)

print(missing\_data)

## ****Covariance Matrix. Variability comparison between categories of variables****

corrmat = train\_df.corr()

f, ax = plt.subplots(figsize=(12, 9))

sns.heatmap(corrmat, vmax=.8, square=True);

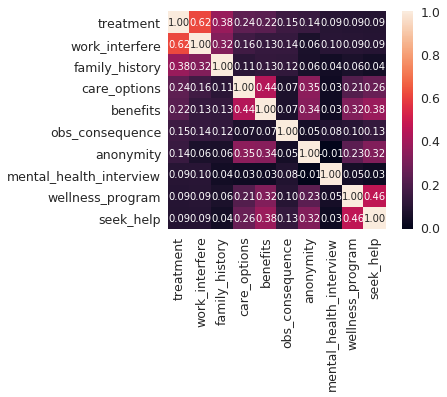
plt.show()

k = 10 *#number of variables for heatmap*

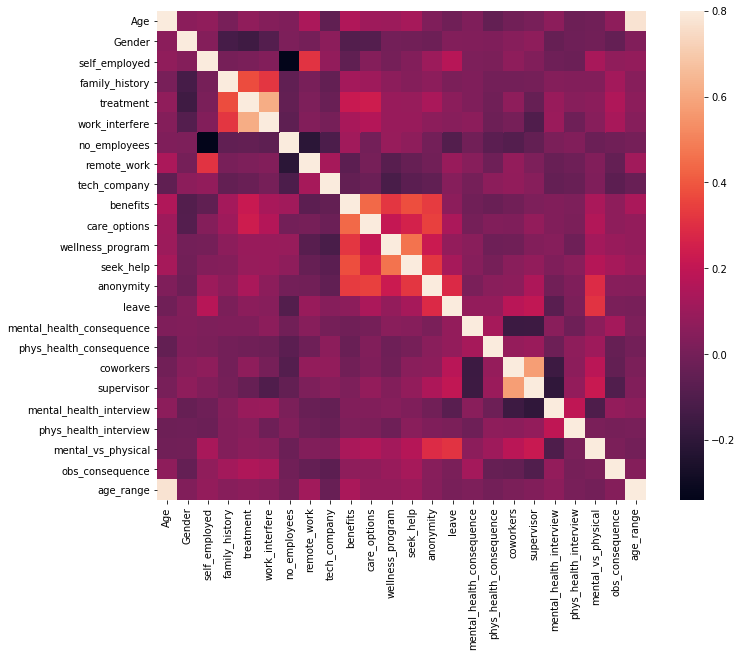
cols = corrmat.nlargest(k, 'treatment')['treatment'].index

cm = np.corrcoef(train\_df[cols].values.T)

sns.set(font\_scale=1.25)

hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot\_kws={'size': 10}, yticklabels=cols.values, xticklabels=cols.values)

plt.show()



## ****Some charts to see data relationship****

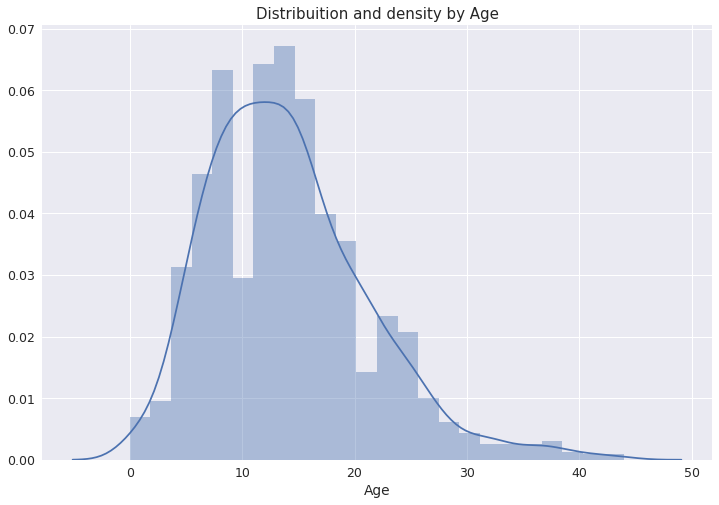
**Distribiution and density by Age**

plt.figure(figsize=(12,8))

sns.distplot(train\_df["Age"], bins=24)

plt.title("Distribuition and density by Age")

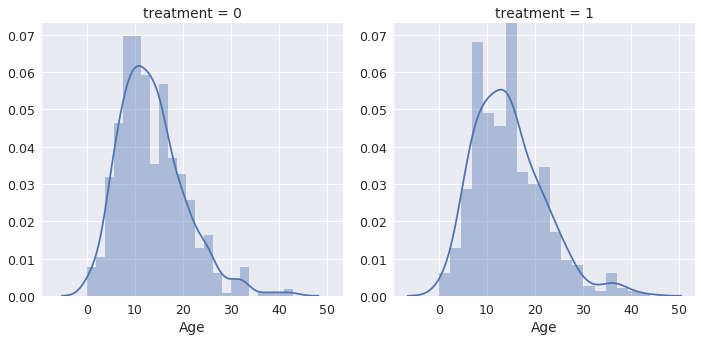
plt.xlabel("Age")



**Separate by treatment**

g = sns.FacetGrid(train\_df, col='treatment', size=5)

g = g.map(sns.distplot, "Age")



**How many people has been treated?**

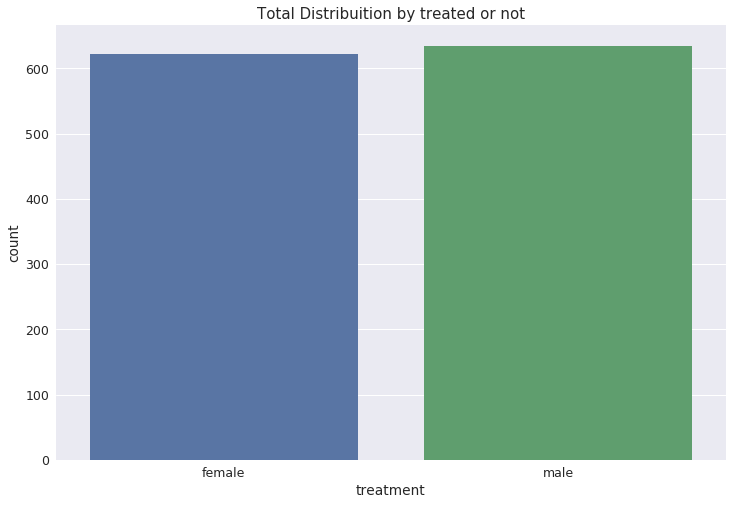
plt.figure(figsize=(12,8))

labels = labelDict['label\_Gender']

g = sns.countplot(x="treatment", data=train\_df)

g.set\_xticklabels(labels)

plt.title('Total Distribuition by treated or not')

****

**Draw a nested barplot to show probabilities for class and sex**

o = labelDict['label\_age\_range']

g = sns.factorplot(x="age\_range", y="treatment", hue="Gender", data=train\_df, kind="bar", ci=None, size=5, aspect=2, legend\_out = True)

g.set\_xticklabels(o)

plt.title('Probability of mental health condition')

plt.ylabel('Probability x 100')

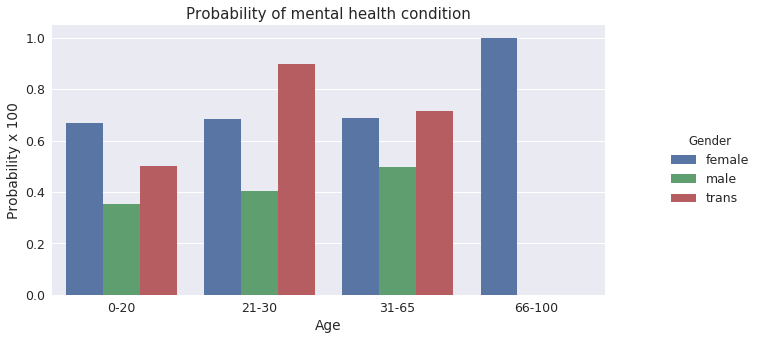
plt.xlabel('Age')

new\_labels = labelDict['label\_Gender']

for t, l **in** zip(g.\_legend.texts, new\_labels): t.set\_text(l)

g.fig.subplots\_adjust(top=0.9,right=0.8)

plt.show()

****

**Barplot to show probabilities for family history**

o = labelDict['label\_family\_history']

g = sns.factorplot(x="family\_history", y="treatment", hue="Gender", data=train\_df, kind="bar", ci=None, size=5, aspect=2, legend\_out = True)

g.set\_xticklabels(o)

plt.title('Probability of mental health condition')

plt.ylabel('Probability x 100')

plt.xlabel('Family History')

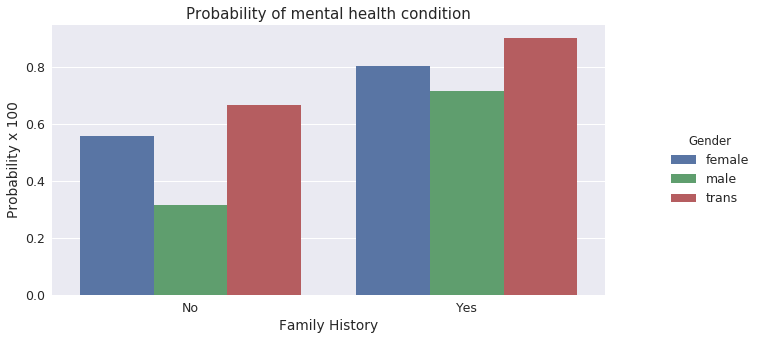
*ls*

new\_labels = labelDict['label\_Gender']

for t, l **in** zip(g.\_legend.texts, new\_labels): t.set\_text(l)

g.fig.subplots\_adjust(top=0.9,right=0.8)

plt.show()



**Barplot to show probabilities for care options**

o = labelDict['label\_care\_options']

g = sns.factorplot(x="care\_options", y="treatment", hue="Gender", data=train\_df, kind="bar", ci=None, size=5, aspect=2, legend\_out = True)

g.set\_xticklabels(o)

plt.title('Probability of mental health condition')

plt.ylabel('Probability x 100')

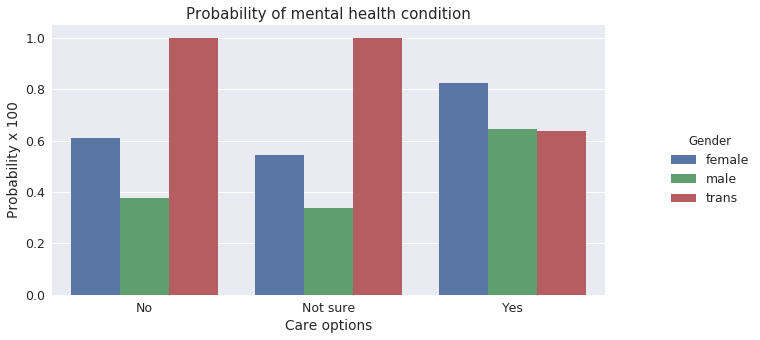
plt.xlabel('Care options')

new\_labels = labelDict['label\_Gender']

for t, l **in** zip(g.\_legend.texts, new\_labels): t.set\_text(l)

g.fig.subplots\_adjust(top=0.9,right=0.8)

plt.show()



**Barplot to show probabilities for benefits**

In [18]:

o = labelDict['label\_benefits']

g = sns.factorplot(x="care\_options", y="treatment", hue="Gender", data=train\_df, kind="bar", ci=None, size=5, aspect=2, legend\_out = True)

g.set\_xticklabels(o)

plt.title('Probability of mental health condition')

plt.ylabel('Probability x 100')

plt.xlabel('Benefits')

*# replace legend labels*

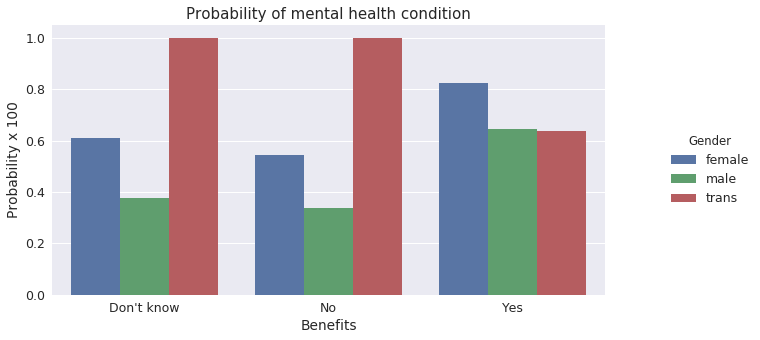
new\_labels = labelDict['label\_Gender']

for t, l **in** zip(g.\_legend.texts, new\_labels): t.set\_text(l)

*# Positioning the legend*

g.fig.subplots\_adjust(top=0.9,right=0.8)

plt.show()



**Barplot to show probabilities for work interfere**

In [19]:

o = labelDict['label\_work\_interfere']

g = sns.factorplot(x="work\_interfere", y="treatment", hue="Gender", data=train\_df, kind="bar", ci=None, size=5, aspect=2, legend\_out = True)

g.set\_xticklabels(o)

plt.title('Probability of mental health condition')

plt.ylabel('Probability x 100')

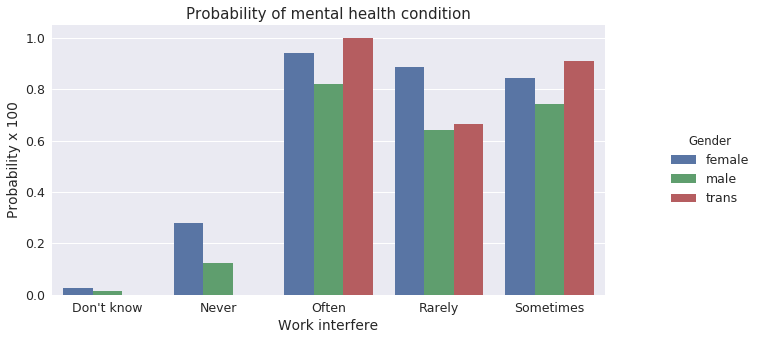
plt.xlabel('Work interfere')

new\_labels = labelDict['label\_Gender']

for t, l **in** zip(g.\_legend.texts, new\_labels): t.set\_text(l)

g.fig.subplots\_adjust(top=0.9,right=0.8)

plt.show()



## ****Scaling and fitting****

scaler = MinMaxScaler()

train\_df['Age'] = scaler.fit\_transform(train\_df[['Age']])

train\_df.head()

**Spliltting the dataset**

feature\_cols = ['Age', 'Gender', 'family\_history', 'benefits', 'care\_options', 'anonymity', 'leave', 'work\_interfere']

X = train\_df[feature\_cols]

y = train\_df.treatment

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, random\_state=0)

methodDict = {}

rmseDict = ()

forest = ExtraTreesClassifier(n\_estimators=250, random\_state=0)

forest.fit(X, y)

importances = forest.feature\_importances\_

std = np.std([tree.feature\_importances\_ for tree **in** forest.estimators\_],

axis=0)

indices = np.argsort(importances)[::-1]

labels = []

for f **in** range(X.shape[1]):

labels.append(feature\_cols[f])

plt.figure(figsize=(12,8))

plt.title("Feature importances")

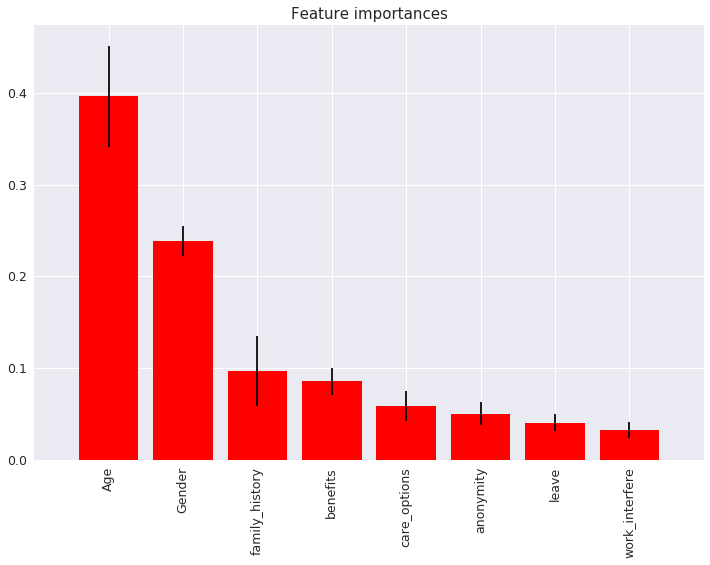
plt.bar(range(X.shape[1]), importances[indices],

color="r", yerr=std[indices], align="center")

plt.xticks(range(X.shape[1]), labels, rotation='vertical')

plt.xlim([-1, X.shape[1]])

plt.show()

****

## ****Tuning****

### ****Evaluating a Classification Model.****

This function will evalue:

* **Classification accuracy:**percentage of correct predictions
* **Null accuracy:** accuracy that could be achieved by always predicting the most frequent class
* **Percentage of ones**
* **Percentage of zero**s
* **Confusion matrix:**Table that describes the performance of a classification model
* True Positives (TP): we correctly predicted that they do have diabetes
* True Negatives (TN): we correctly predicted that they don't have diabetes
* False Positives (FP): we incorrectly predicted that they do have diabetes (a "Type I error")
* Falsely predict positive
* False Negatives (FN): we incorrectly predicted that they don't have diabetes (a "Type II error")

Falsely predict negative

* **False Positive Rate**
* **Precision of Positive value**
* **AUC:** is the percentage of the ROC plot that is underneath the curve
* .90-1 = excellent (A)
* .80-.90 = good (B)
* .70-.80 = fair (C)
* .60-.70 = poor (D)
* .50-.60 = fail (F)

And some others values for tuning processes. More information: [<http://www.ritchieng.com/machine-learning-evaluate-classification-model/>]:

def evalClassModel(model, y\_test, y\_pred\_class, plot=False):

print('Accuracy:', metrics.accuracy\_score(y\_test, y\_pred\_class))

print('Null accuracy:**\n**', y\_test.value\_counts())

print('Percentage of ones:', y\_test.mean())

print('Percentage of zeros:',1 - y\_test.mean())

print('True:', y\_test.values[0:25])

print('Pred:', y\_pred\_class[0:25])

TP = confusion[1, 1]

TN = confusion[0, 0]

FP = confusion[0, 1]

FN = confusion[1, 0]

sns.heatmap(confusion,annot=True,fmt="d")

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.show()

accuracy = metrics.accuracy\_score(y\_test, y\_pred\_class)

print('Classification Accuracy:', accuracy)

print('Classification Error:', 1 - metrics.accuracy\_score(y\_test, y\_pred\_class))

false\_positive\_rate = FP / float(TN + FP)

print('False Positive Rate:', false\_positive\_rate)

print('Precision:', metrics.precision\_score(y\_test, y\_pred\_class))

print('AUC Score:', metrics.roc\_auc\_score(y\_test, y\_pred\_class))

print('Cross-validated AUC:', cross\_val\_score(model, X, y, cv=10, scoring='roc\_auc').mean())

print('First 10 predicted responses:**\n**', model.predict(X\_test)[0:10])

print('First 10 predicted probabilities of class members:**\n**', model.predict\_proba(X\_test)[0:10])

model.predict\_proba(X\_test)[0:10, 1]

y\_pred\_prob = model.predict\_proba(X\_test)[:, 1]

if plot == True:

plt.rcParams['font.size'] = 12

plt.hist(y\_pred\_prob, bins=8)

plt.xlim(0,1)

plt.title('Histogram of predicted probabilities')

plt.xlabel('Predicted probability of treatment')

plt.ylabel('Frequency')

y\_pred\_prob = y\_pred\_prob.reshape(-1,1)

y\_pred\_class = binarize(y\_pred\_prob, 0.3)[0]

print('First 10 predicted probabilities:**\n**', y\_pred\_prob[0:10])

roc\_auc = metrics.roc\_auc\_score(y\_test, y\_pred\_prob)

fpr, tpr, thresholds = metrics.roc\_curve(y\_test, y\_pred\_prob)

if plot == True:

plt.figure()

plt.plot(fpr, tpr, color='darkorange', label='ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color='navy', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.0])

plt.rcParams['font.size'] = 12

plt.title('ROC curve for treatment classifier')

plt.xlabel('False Positive Rate (1 - Specificity)')

plt.ylabel('True Positive Rate (Sensitivity)')

plt.legend(loc="lower right")

plt.show()

def evaluate\_threshold(threshold):

print('Specificity for ' + str(threshold) + ' :', 1 - fpr[thresholds > threshold][-1])

predict\_mine = np.where(y\_pred\_prob > 0.50, 1, 0)

confusion = metrics.confusion\_matrix(y\_test, predict\_mine)

print(confusion)

return accuracy

### ****Tuning with cross validation score****

def tuningCV(knn):

k\_range = list(range(1, 31))

k\_scores = []

for k **in** k\_range:

knn = KNeighborsClassifier(n\_neighbors=k)

scores = cross\_val\_score(knn, X, y, cv=10, scoring='accuracy')

k\_scores.append(scores.mean())

print(k\_scores)

plt.plot(k\_range, k\_scores)

plt.xlabel('Value of K for KNN')

plt.ylabel('Cross-Validated Accuracy')

plt.show()

### ****Tuning with GridSearchCV****

def tuningGridSerach(knn):

k\_range = list(range(1, 31))

print(k\_range)

param\_grid = dict(n\_neighbors=k\_range)

print(param\_grid)

grid = GridSearchCV(knn, param\_grid, cv=10, scoring='accuracy')

grid.grid\_scores\_

print(grid.grid\_scores\_[0].parameters)

print(grid.grid\_scores\_[0].cv\_validation\_scores)

print(grid.grid\_scores\_[0].mean\_validation\_score)

grid\_mean\_scores = [result.mean\_validation\_score for result **in** grid.grid\_scores\_]

print(grid\_mean\_scores)

plt.plot(k\_range, grid\_mean\_scores)

plt.xlabel('Value of K for KNN')

plt.ylabel('Cross-Validated Accuracy')

plt.show()

print('GridSearch best score', grid.best\_score\_)

print('GridSearch best params', grid.best\_params\_)

print('GridSearch best estimator', grid.best\_estimator\_)

### ****Tuning with RandomizedSearchCV****

def tuningRandomizedSearchCV(model, param\_dist):

rand = RandomizedSearchCV(model, param\_dist, cv=10, scoring='accuracy', n\_iter=10, random\_state=5)

rand.fit(X, y)

rand.grid\_scores\_

print('Rand. Best Score: ', rand.best\_score\_)

print('Rand. Best Params: ', rand.best\_params\_)

best\_scores = []

for \_ **in** range(20):

rand = RandomizedSearchCV(model, param\_dist, cv=10, scoring='accuracy', n\_iter=10)

rand.fit(X, y)

best\_scores.append(round(rand.best\_score\_, 3))

print(best\_scores)

### ****Tuning with searching multiple parameters simultaneously****

def tuningMultParam(knn):

k\_range = list(range(1, 31))

weight\_options = ['uniform', 'distance']

param\_grid = dict(n\_neighbors=k\_range, weights=weight\_options)

print(param\_grid)

grid = GridSearchCV(knn, param\_grid, cv=10, scoring='accuracy')

grid.fit(X, y)

print(grid.grid\_scores\_)

print('Multiparam. Best Score: ', grid.best\_score\_)

print('Multiparam. Best Params: ', grid.best\_params\_)

## Random Forests

def randomForest():

forest = RandomForestClassifier(n\_estimators = 20)

featuresSize = feature\_cols.\_\_len\_\_()

param\_dist = {"max\_depth": [3, None],

"max\_features": randint(1, featuresSize),

"min\_samples\_split": randint(2, 9),

"min\_samples\_leaf": randint(1, 9),

"criterion": ["gini", "entropy"]}

tuningRandomizedSearchCV(forest, param\_dist)

forest = RandomForestClassifier(max\_depth = None, min\_samples\_leaf=8, min\_samples\_split=2, n\_estimators = 20, random\_state = 1)

my\_forest = forest.fit(X\_train, y\_train)

y\_pred\_class = my\_forest.predict(X\_test)

print('########### Random Forests ###############')

accuracy\_score = evalClassModel(my\_forest, y\_test, y\_pred\_class, True)

methodDict['R. Forest'] = accuracy\_score \* 100

randomForest()

Rand. Best Score: 0.8329355608591885

Rand. Best Params: {'criterion': 'gini', 'max\_depth': 3, 'max\_features': 3, 'min\_samples\_leaf': 8, 'min\_samples\_split': 6}

[0.835, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.833, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831, 0.831]

########### Random Forests ###############

Accuracy: 0.812169312169

Null accuracy:

0 191

1 187

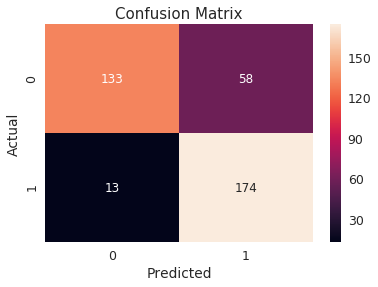
Name: treatment, dtype: int64

Percentage of ones: 0.4947089947089947

Percentage of zeros: 0.5052910052910053

True: [0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0]

Pred: [1 0 0 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0]



Classification Accuracy: 0.812169312169

Classification Error: 0.187830687831

False Positive Rate: 0.303664921466

Precision: 0.75

AUC Score: 0.813408180978

Cross-validated AUC: 0.893237767217

First 10 predicted responses:

[1 0 0 0 1 1 0 1 1 1]

First 10 predicted probabilities of class members:

[[ 0.2555794 0.7444206 ]

[ 0.95069083 0.04930917]

[ 0.93851009 0.06148991]

[ 0.87096597 0.12903403]

[ 0.40653554 0.59346446]

[ 0.17282958 0.82717042]

[ 0.89450448 0.10549552]

[ 0.4065912 0.5934088 ]

[ 0.20540631 0.79459369]

[ 0.19337644 0.80662356]]

First 10 predicted probabilities:

[[ 0.7444206 ]

[ 0.04930917]

[ 0.06148991]

[ 0.12903403]

[ 0.59346446]

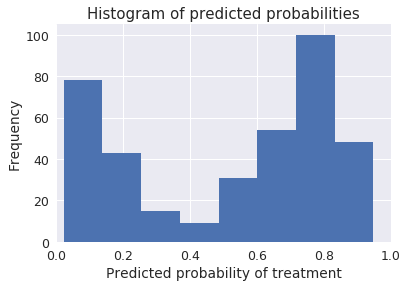
[ 0.82717042]

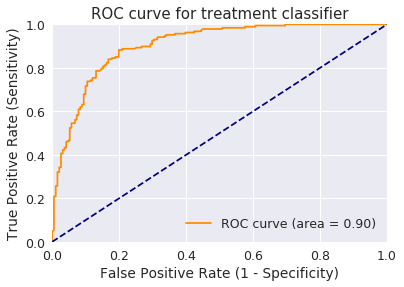
[ 0.10549552]

[ 0.5934088 ]

[ 0.79459369]

[ 0.80662356]]





[[133 58]

[ 13 174]]

### Bagging

def bagging():

bag = BaggingClassifier(DecisionTreeClassifier(), max\_samples=1.0, max\_features=1.0, bootstrap\_features=False)

bag.fit(X\_train, y\_train)

y\_pred\_class = bag.predict(X\_test)

print('########### Bagging ###############')

accuracy\_score = evalClassModel(bag, y\_test, y\_pred\_class, True)

methodDict['Bagging'] = accuracy\_score \* 100

########### Bagging ###############

Accuracy: 0.785714285714

Null accuracy:

0 191

1 187

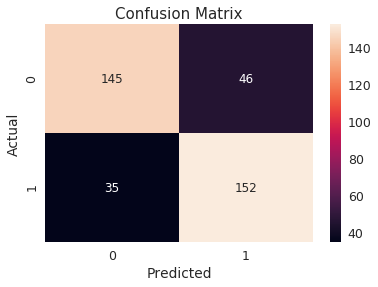
Name: treatment, dtype: int64

Percentage of ones: 0.4947089947089947

Percentage of zeros: 0.5052910052910053

True: [0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0]

Pred: [1 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 0 0 0 0 1 0 0]



Classification Accuracy: 0.785714285714

Classification Error: 0.214285714286

False Positive Rate: 0.240837696335

Precision: 0.767676767677

AUC Score: 0.785998264132

Cross-validated AUC: 0.847359030978

First 10 predicted responses:

[1 0 0 0 0 1 0 0 1 1]

First 10 predicted probabilities of class members:

[[ 0.44 0.56]

[ 1. 0. ]

[ 1. 0. ]

[ 0.9 0.1 ]

[ 0.8 0.2 ]

[ 0.3 0.7 ]

[ 1. 0. ]

[ 0.7 0.3 ]

[ 0.2 0.8 ]

[ 0. 1. ]]

First 10 predicted probabilities:

[[ 0.56]

[ 0. ]

[ 0. ]

[ 0.1 ]

[ 0.2 ]

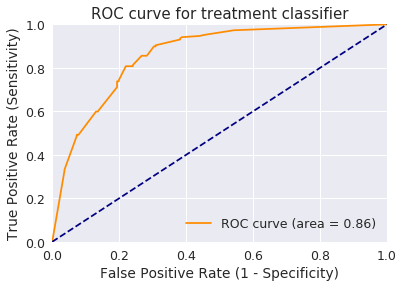
[ 0.7 ]

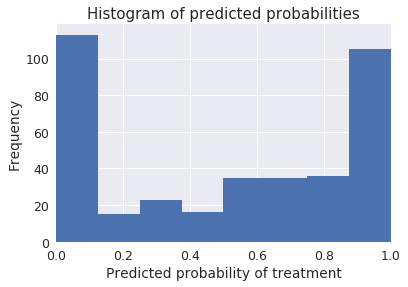
[ 0. ]

[ 0.3 ]

[ 0.8 ]

[ 1. ]]





[[145 46]

[ 35 152]]

### Boosting

def boosting():

clf = DecisionTreeClassifier(criterion='entropy', max\_depth=1)

boost = AdaBoostClassifier(base\_estimator=clf, n\_estimators=500)

boost.fit(X\_train, y\_train)

y\_pred\_class = boost.predict(X\_test)

print('########### Boosting ###############')

accuracy\_score = evalClassModel(boost, y\_test, y\_pred\_class, True)

methodDict['Boosting'] = accuracy\_score \* 100

boosting()

########### Boosting ###############

Accuracy: 0.81746031746

Null accuracy:

0 191

1 187

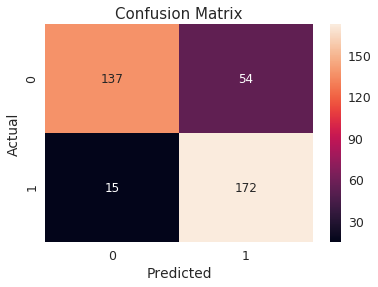
Name: treatment, dtype: int64

Percentage of ones: 0.4947089947089947

Percentage of zeros: 0.5052910052910053

True: [0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0]

Pred: [1 0 0 0 0 1 0 1 1 1 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0]



Classification Accuracy: 0.81746031746

Classification Error: 0.18253968254

False Positive Rate: 0.282722513089

Precision: 0.761061946903

AUC Score: 0.818531791584

Cross-validated AUC: 0.874085341462

First 10 predicted responses:

[1 0 0 0 0 1 0 1 1 1]

First 10 predicted probabilities of class members:

[[ 0.49924555 0.50075445]

[ 0.50285507 0.49714493]

[ 0.50291786 0.49708214]

[ 0.50127788 0.49872212]

[ 0.50013552 0.49986448]

[ 0.49796157 0.50203843]

[ 0.50046371 0.49953629]

[ 0.49939483 0.50060517]

[ 0.49921757 0.50078243]

[ 0.49897133 0.50102867]]

First 10 predicted probabilities:

[[ 0.50075445]

[ 0.49714493]

[ 0.49708214]

[ 0.49872212]

[ 0.49986448]

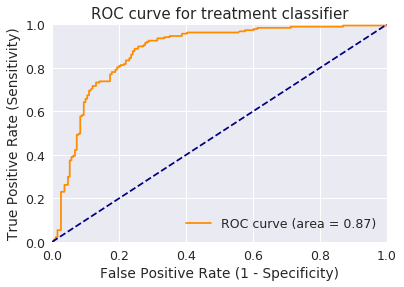
[ 0.50203843]

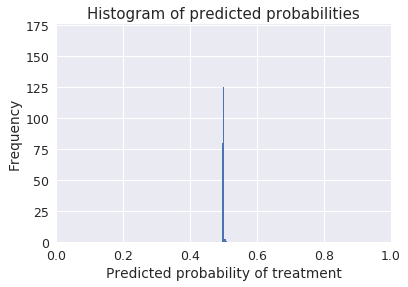
[ 0.49953629]

[ 0.50060517]

[ 0.50078243]

[ 0.50102867]]





[[137 54]

[ 15 172]]

### Stacking

def stacking():

clf1 = KNeighborsClassifier(n\_neighbors=1)

clf2 = RandomForestClassifier(random\_state=1)

clf3 = GaussianNB()

lr = LogisticRegression()

stack = StackingClassifier(classifiers=[clf1, clf2, clf3], meta\_classifier=lr)

stack.fit(X\_train, y\_train)

y\_pred\_class = stack.predict(X\_test)

print('########### Stacking ###############')

accuracy\_score = evalClassModel(stack, y\_test, y\_pred\_class, True)

methodDict['Stacking'] = accuracy\_score \* 100

stacking()

########### Stacking ###############

Accuracy: 0.759259259259

Null accuracy:

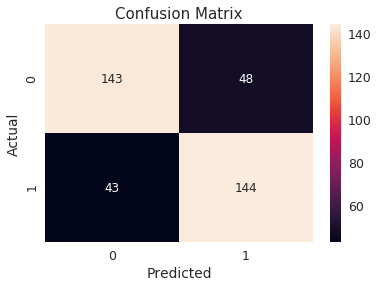
0 191

1 187

Name: treatment, dtype: int64

Percentage of ones: 0.4947089947089947

Percentage of zeros: 0.5052910052910053



True: [0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0]

Pred: [1 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 0 0 0 0 1 0 0]

Classification Accuracy: 0.759259259259

Classification Error: 0.240740740741

False Positive Rate: 0.251308900524

Precision: 0.75

AUC Score: 0.759372287706

Cross-validated AUC: 0.840591797875

First 10 predicted responses:

[1 0 0 0 0 1 0 0 1 1]

First 10 predicted probabilities of class members:

[[ 0.01481486 0.98518514]

[ 0.98086439 0.01913561]

[ 0.98086439 0.01913561]

[ 0.98086439 0.01913561]

[ 0.98086439 0.01913561]

[ 0.01481486 0.98518514]

[ 0.98086439 0.01913561]

[ 0.9602071 0.0397929 ]

[ 0.030955 0.969045 ]

[ 0.01481486 0.98518514]]

First 10 predicted probabilities:

[[ 0.98518514]

[ 0.01913561]

[ 0.01913561]

[ 0.01913561]

[ 0.01913561]

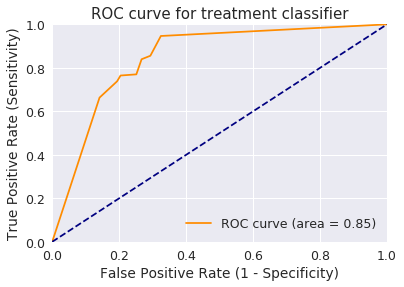
[ 0.98518514]

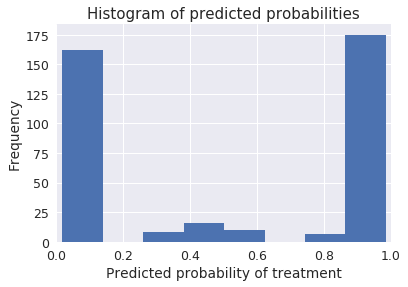
[ 0.01913561]

[ 0.0397929 ]

[ 0.969045 ]

[ 0.98518514]]





[[143 48]

[ 43 144]]

## ****Success method plot****

def plotSuccess():

s = pd.Series(methodDict)

s = s.sort\_values(ascending=False)

plt.figure(figsize=(12,8))

ax = s.plot(kind='bar')

for p **in** ax.patches:

ax.annotate(str(round(p.get\_height(),2)), (p.get\_x() \* 1.005, p.get\_height() \* 1.005))

plt.ylim([70.0, 90.0])

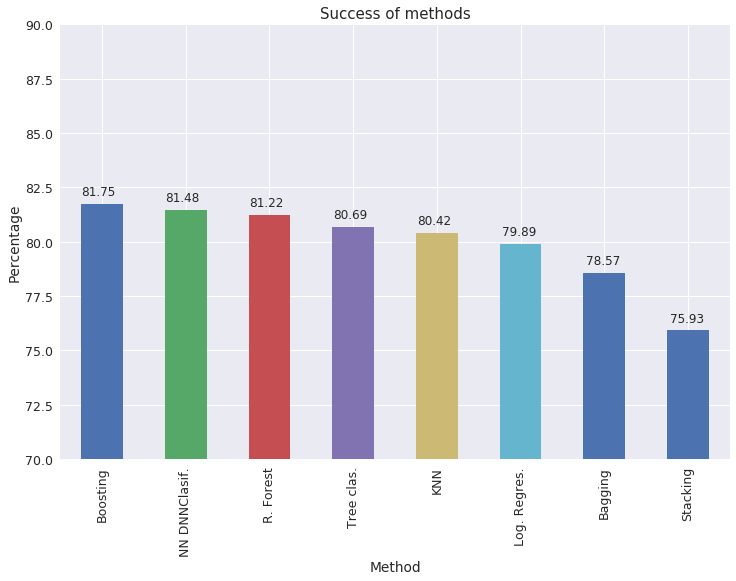
plt.xlabel('Method')

plt.ylabel('Percentage')

plt.title('Success of methods')

plt.show()

plotSuccess()



## ****Creating predictions on test set****

clf = AdaBoostClassifier()

clf.fit(X, y)

dfTestPredictions = clf.predict(X\_test)

results = pd.DataFrame({'Index': X\_test.index, 'Treatment': dfTestPredictions})

results.to\_csv('results.csv', index=False)

results.head()

|  | Index | Treatment |
| --- | --- | --- |
| 0 | 5 | 1 |
| 1 | 494 | 0 |
| 2 | 52 | 0 |
| 3 | 984 | 0 |
| 4 | 186 | 0 |

## ****Conclusions****

As a beginner I don't know whether the results are the best. I think over 80% of success in the majority of methods is a good rate, given the point is to know whether a patient needs treatment or not.

There's only left to have a way to persist the model for future use without having to retrain. This job will be done in another kernel.

Thanks for reading and if you'd like my job or want to give some advice, feel free.