JAYPEE INSTITUTE OF INFORMATION AND TECHNOLOGY

OPEN SOURCE SOFTWARE

PROJECT REPORT

TOPIC:

GROUP DETAILS:

NAME	ENROLLMENT NO.	BATCH
Riya kansal	20103251	B9
Geetali Agarwal	20103254	B9
Kritarth bansal	20103256	B9
Saksham gupta	20103268	B9

FACULTY- Mrs . Purtee Kohli

Abstract

The increase of mental health problems and the need for effective medical health care have led to an investigation of machine learning that can be applied in mental health problems. This paper presents a recent systematic review of machine learning approaches in predicting mental health problems. Furthermore, we will discuss the challenges, limitations, and future directions for the application of machine learning in the mental health field. We collect research articles and studies that are related to the machine learning approaches in predicting mental health problems by searching reliable databases. Moreover, we adhere to the PRISMA methodology in conducting this systematic review. We include a total of 30 research articles in this review after the screening and identification processes. Then, we categorize the collected research articles based on the mental health problems such as schizophrenia, bipolar disorder, anxiety and depression, posttraumatic stress disorder, and mental health problems among children. Discussing the findings, we reflect on the challenges and limitations faced by the researchers on machine learning in mental health problems. Additionally, we provide concrete recommendations on the potential future research and development of applying machine learning in the mental health field.

Introduction

Mental illness is a health problem that undoubtedly impacts emotions, reasoning, and social interaction of a person. These issues have shown that mental illness gives serious consequences across societies and demands new strategies for prevention and intervention. To accomplish these strategies, early detection of mental health is an essential procedure. Medical predictive analytics will reform the healthcare field broadly as discussed by Miner et al. [1]. Mental illness is usually diagnosed based on the individual self-report that requires questionnaires designed for the detection of the specific patterns of feeling or social interactions [2]. With proper care and treatment, many individuals will hopefully be able to recover from mental illness or emotional disorder

Machine learning is a technique that aims to construct systems that can improve through experience by using advanced statistical and probabilistic techniques. It is believed to be a significantly useful tool to help in predicting mental health. It is allowing many researchers to acquire important information from the data, provide personalized experiences, and develop automated intelligent systems [4]. The widely used algorithms in the field of machine learning such as support vector machine, random forest, and artificial neural networks have been utilized to forecast and categorize the future events

METHODOLOGY

1. About the dataset
2. Load essential Python Libraries
3. Load Training/Test datasets
4. Data Preprocessing
5. Exploratory data analysis (EDA).
6. Feature Engineering.
7. Build Machine Learning Model
8. Make predictions on the test dataset

OBJECTIVE

This review aimed to provide a concise snapshot of the research to date investigating ML applications to mental health. Previous reviews have demonstrated ML techniques to be robust and scalable for mental health application, but no review has comprehensively mapped the clinical applications within mental health research and practice. Such a review would equip both data scientists and practitioners in the methods and applications of big data. It would also highlight the challenges of using ML techniques in this context, as well as identify gaps in the field and potential opportunities for further research. First, we outline the search strategies used to find relevant literature. Next, we conduct a synthesis of the literature, describing both the ML techniques and mental health applications of each article. Finally, the paper summarises the extant research and the implications for future work.

CONCLUSION :

Above is the report of the project with data preprocessing , training and test data , feature engineering etc .We have used logistic regression,k neighbour,dicission tree classification based algorithm .accuracy of all algos comes out to be 79.63%,80.42%,80.69%

CODE AND OUTPUT:

In [182…
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [183…
```
from scipy import stats
from scipy.stats import randint
```

In [184…
```
# prep
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
```

In [185…
```
# models
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.model_selection import RandomizedSearchCV
```

In [186…
```
# Validation libraries
from sklearn import metrics
from sklearn.metrics import accuracy_score, mean_squared_error, precision_recall_cu
from sklearn.model_selection import cross_val_score
```

In [187…
```
#Bagging
from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
```

In [188…
```
#Naive bayes
from sklearn.naive_bayes import GaussianNB
```

In [189…
```
#Stacking
from mlxtend.classifier import StackingClassifier
```

In [190…
```
train_df = pd.read_csv('survey.csv')
train_df.head()
```

Out[190]:

	Timestamp	Age	Gender	Country	state	self_employed	family_history	treatment	work_int
0	2014-08-27 11:29:31	37	Female	United States	IL	NaN	No	Yes	
1	2014-08-27 11:29:37	44	M	United States	IN	NaN	No	No	
2	2014-08-27 11:29:44	32	Male	Canada	NaN	NaN	No	No	
3	2014-08-27 11:29:46	31	Male	United Kingdom	NaN	NaN	Yes	Yes	
4	2014-08-27 11:30:22	31	Male	United States	TX	NaN	No	No	

5 rows × 27 columns

In [191…
```
#missing data
```

```
total = train_df.isnull().sum().sort_values(ascending=False)
percent = (train_df.isnull().sum()/train_df.isnull().count()).sort_values(ascending
missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
missing_data.head(20)
print(missing_data)
```

```
                           Total   Percent
comments                    1095  0.869738
state                        515  0.409055
work_interfere               264  0.209690
self_employed                 18  0.014297
seek_help                      0  0.000000
obs_consequence                0  0.000000
mental_vs_physical             0  0.000000
phys_health_interview          0  0.000000
mental_health_interview        0  0.000000
supervisor                     0  0.000000
coworkers                      0  0.000000
phys_health_consequence        0  0.000000
mental_health_consequence      0  0.000000
leave                          0  0.000000
anonymity                      0  0.000000
Timestamp                      0  0.000000
wellness_program               0  0.000000
Age                            0  0.000000
benefits                       0  0.000000
tech_company                   0  0.000000
remote_work                    0  0.000000
no_employees                   0  0.000000
treatment                      0  0.000000
family_history                 0  0.000000
Country                        0  0.000000
Gender                         0  0.000000
care_options                   0  0.000000
```

In [192…

```
#dealing with missing data
train_df.drop(['comments'], axis= 1, inplace=True)
train_df.drop(['state'], axis= 1, inplace=True)
train_df.drop(['Timestamp'], axis= 1, inplace=True)
train_df.drop(['Country'], axis= 1, inplace=True)
```

In [193…

```
# Assign default values for each data type
defaultInt = 0
defaultString = 'NaN'
defaultFloat = 0.0

# Create lists by data tpe
intFeatures = ['Age']
stringFeatures = ['Gender', 'self_employed', 'family_history', 'treatment', 'work_
                 'no_employees', 'remote_work', 'tech_company', 'anonymity', 'leave
                 'phys_health_consequence', 'coworkers', 'supervisor', 'mental_hea
                 'mental_vs_physical', 'obs_consequence', 'benefits', 'care_option
                 'seek_help']
floatFeatures = []

# Clean the NaN's
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()
```

Out[193]:

	Age	Gender	self_employed	family_history	treatment	work_interfere	no_employees	remote
0	37	Female	NaN	No	Yes	Often	6-25	
1	44	M	NaN	No	No	Rarely	More than 1000	
2	32	Male	NaN	No	No	Rarely	6-25	
3	31	Male	NaN	Yes	Yes	Often	26-100	
4	31	Male	NaN	No	No	Never	100-500	

5 rows × 23 columns

In [194…

```
#Clean 'Gender'
gender = train_df['Gender'].unique()
print(gender)
```

```
['Female' 'M' 'Male' 'male' 'female' 'm' 'Male-ish' 'maile' 'Trans-female'
 'Cis Female' 'F' 'something kinda male?' 'Cis Male' 'Woman' 'f' 'Mal'
 'Male (CIS)' 'queer/she/they' 'non-binary' 'Femake' 'woman' 'Make' 'Nah'
 'All' 'Enby' 'fluid' 'Genderqueer' 'Female ' 'Androgyne' 'Agender'
 'cis-female/femme' 'Guy (-ish) ^_^' 'male leaning androgynous' 'Male '
 'Man' 'Trans woman' 'msle' 'Neuter' 'Female (trans)' 'queer'
 'Female (cis)' 'Mail' 'cis male' 'A little about you' 'Malr' 'p' 'femail'
 'Cis Man' 'ostensibly male, unsure what that really means']
```

In [195…

```
#Made gender groups
male_str = ["male", "m", "male-ish", "maile", "mal", "male (cis)", "make", "male ",
trans_str = ["trans-female", "something kinda male?", "queer/she/they", "non-binary
female_str = ["cis female", "f", "female", "woman",  "femake", "female ","cis-femal

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=Tru

    if str.lower(col.Gender) in female_str:
        train_df['Gender'].replace(to_replace=col.Gender, value='female', inplace=

    if str.lower(col.Gender) in trans_str:
        train_df['Gender'].replace(to_replace=col.Gender, value='trans', inplace=Tr

#Get rid of bullshit
stk_list = ['A little about you', 'p']
train_df = train_df[~train_df['Gender'].isin(stk_list)]

print(train_df['Gender'].unique())
```

```
['female' 'male' 'trans']
```

In [196…

```
#complete missing age with mean
train_df['Age'].fillna(train_df['Age'].median(), inplace = True)

# Fill with media() values < 18 and > 120
```

```
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

#Ranges of Age
train_df['age_range'] = pd.cut(train_df['Age'], [0,20,30,65,100], labels=["0-20", '
```

In [197…

```
#There are only 0.014% of self employed so let's change NaN to NOT self_employed
#Replace "NaN" string from defaultString
train_df['self_employed'] = train_df['self_employed'].replace([defaultString], 'No
print(train_df['self_employed'].unique())
```

```
['No' 'Yes']
```

In [198…

```
#There are only 0.20% of self work_interfere so let's change NaN to "Don't know
#Replace "NaN" string from defaultString

train_df['work_interfere'] = train_df['work_interfere'].replace([defaultString], 'D
print(train_df['work_interfere'].unique())
```

```
['Often' 'Rarely' 'Never' 'Sometimes' "Don't know"]
```

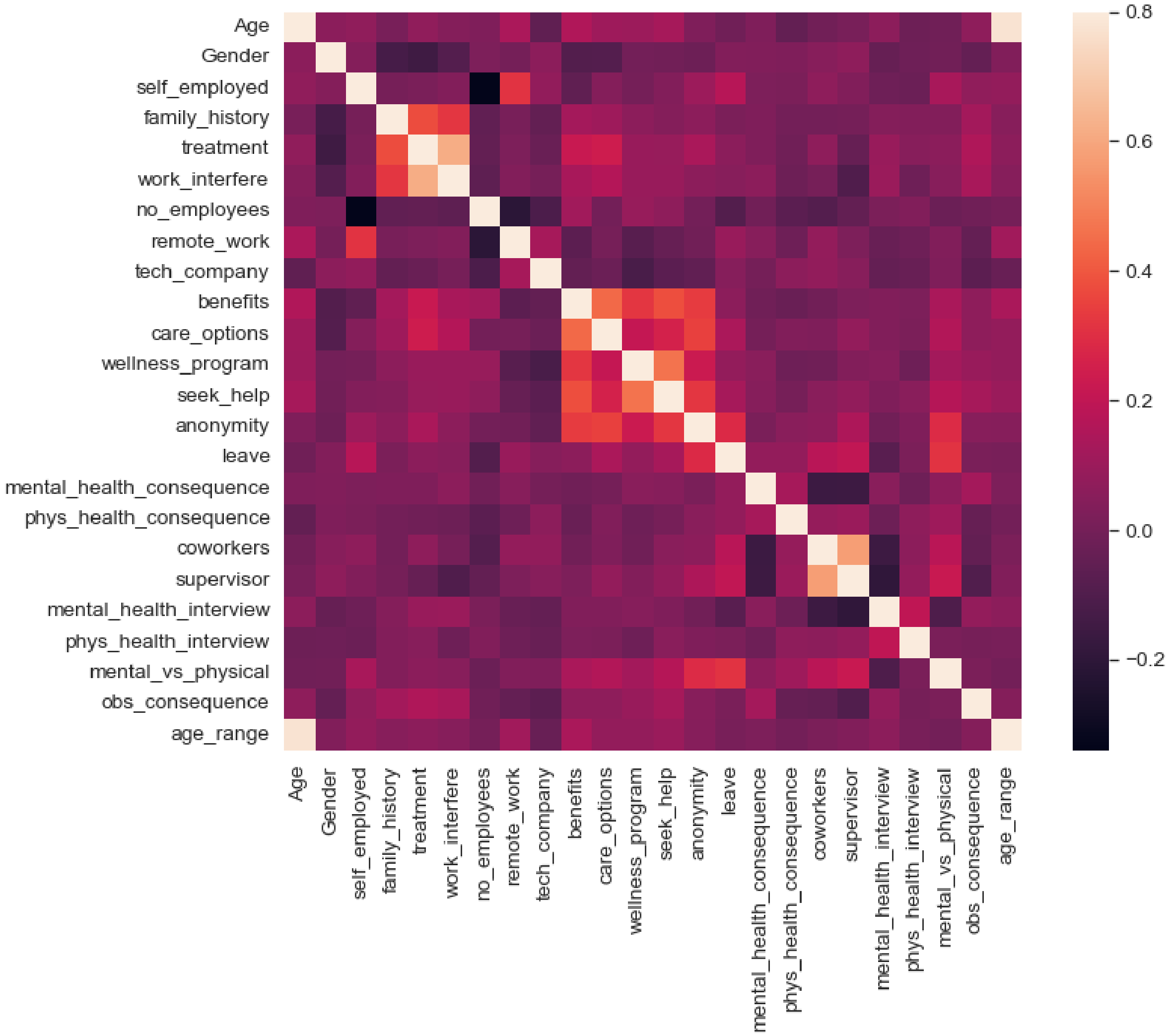
In [199…

```
#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])
    # Get labels
    labelKey = 'label_' + feature
    labelValue = [*le_name_mapping]
    labelDict[labelKey] =labelValue

for key, value in labelDict.items():
    print(key, value)
```

```
label_Age [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 56, 5
7, 58, 60, 61, 62, 65, 72]
label_Gender ['female', 'male', 'trans']
label_self_employed ['No', 'Yes']
label_family_history ['No', 'Yes']
label_treatment ['No', 'Yes']
label_work_interfere ["Don't know", 'Never', 'Often', 'Rarely', 'Sometimes']
label_no_employees ['1-5', '100-500', '26-100', '500-1000', '6-25', 'More than 100
0']
label_remote_work ['No', 'Yes']
label_tech_company ['No', 'Yes']
label_benefits ["Don't know", 'No', 'Yes']
label_care_options ['No', 'Not sure', 'Yes']
label_wellness_program ["Don't know", 'No', 'Yes']
label_seek_help ["Don't know", 'No', 'Yes']
label_anonymity ["Don't know", 'No', 'Yes']
label_leave ["Don't know", 'Somewhat difficult', 'Somewhat easy', 'Very difficul
t', 'Very easy']
label_mental_health_consequence ['Maybe', 'No', 'Yes']
label_phys_health_consequence ['Maybe', 'No', 'Yes']
label_coworkers ['No', 'Some of them', 'Yes']
label_supervisor ['No', 'Some of them', 'Yes']
label_mental_health_interview ['Maybe', 'No', 'Yes']
label_phys_health_interview ['Maybe', 'No', 'Yes']
label_mental_vs_physical ["Don't know", 'No', 'Yes']
label_obs_consequence ['No', 'Yes']
label_age_range ['0-20', '21-30', '31-65', '66-100']
```

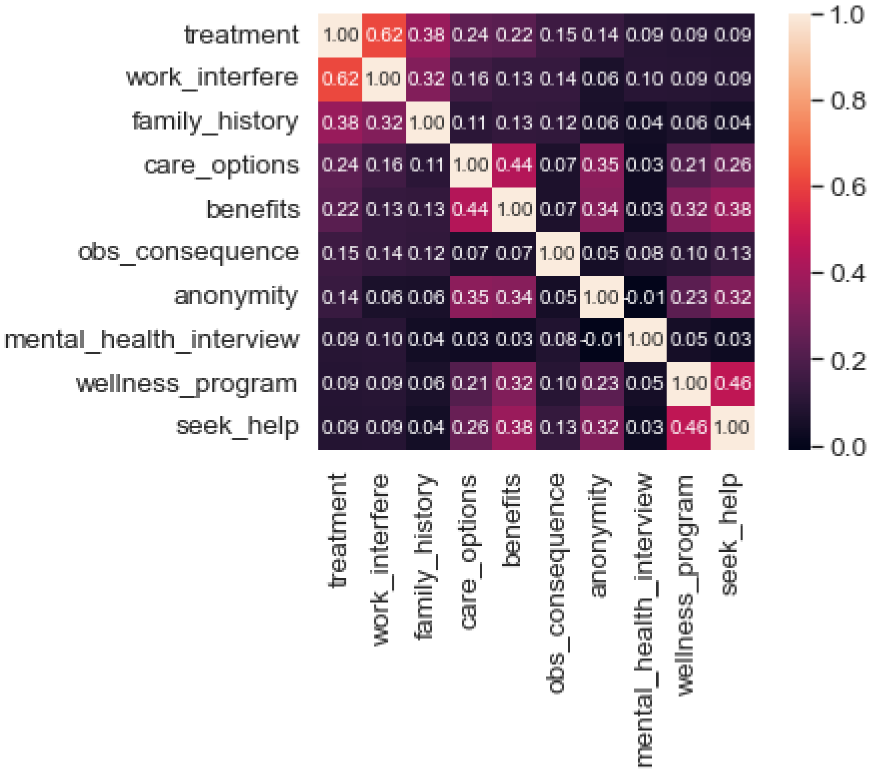
In [200…

```
#correlation matrix
train_df.corr()
f, ax = plt.subplots(figsize=(12, 9))
sns.heatmap(corrmat, vmax=.8, square=True);
plt.show()
```



In [201…

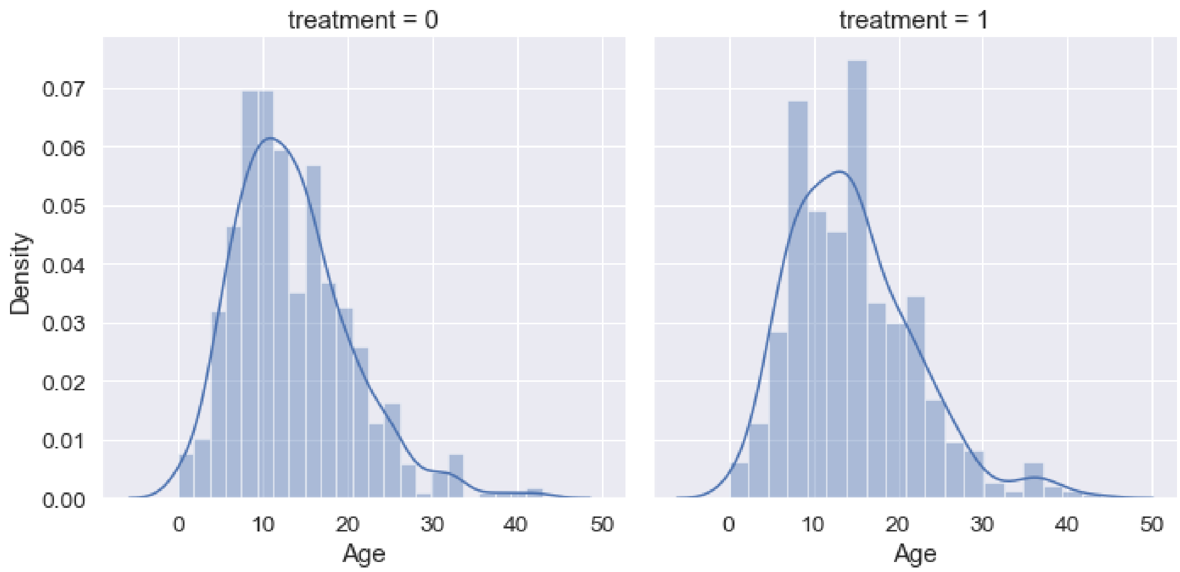
```
#treatment correlation matrix
k = 10 #number of variables for heatmap
cols = corrmat.nlargest(k, 'treatment')['treatment'].index
cm = np.corrcoef(train_df[cols].values.T)
sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'si
plt.show()
```



In [202…

```
g = sns.FacetGrid(train_df, col='treatment', size=5)
g = g.map(sns.distplot, "Age")
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\axisgrid.py:337: UserWarning: T
he `size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWa
rning: `distplot` is a deprecated function and will be removed in a future versio
n. Please adapt your code to use either `displot` (a figure-level function with si
milar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWa
rning: `distplot` is a deprecated function and will be removed in a future versio
n. Please adapt your code to use either `displot` (a figure-level function with si
milar flexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

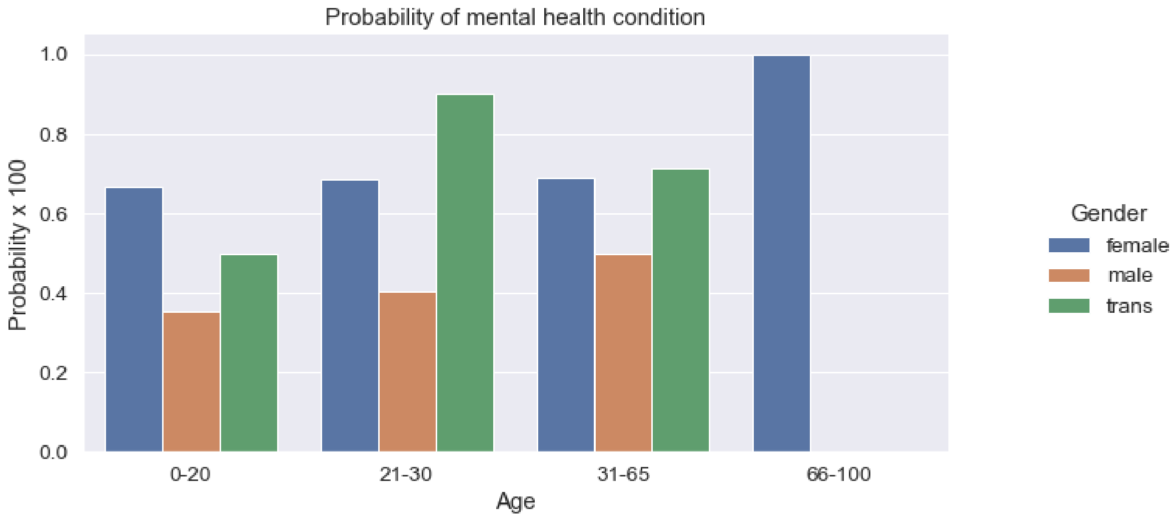


In [203…

```
o = labelDict['label_age_range']

g = sns.factorplot(x="age_range", y="treatment", hue="Gender", data=train_df, kind
```

```
g.set_xticklabels(o)

plt.title('Probability of mental health condition')
plt.ylabel('Probability x 100')
plt.xlabel('Age')
# 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()
```

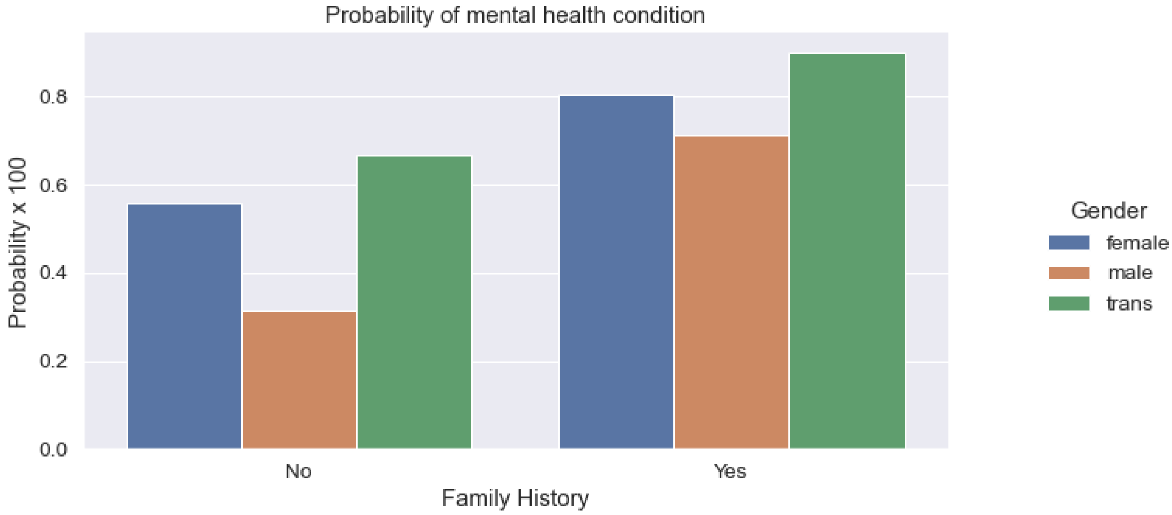
```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\categorical.py:3717: UserWarnin
g: The `factorplot` function has been renamed to `catplot`. The original name will
be removed in a future release. Please update your code. Note that the default `ki
nd` in `factorplot` (`'point'`) has changed `'strip'` in `catplot`.
  warnings.warn(msg)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\categorical.py:3723: UserWarnin
g: The `size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```



In [204…

```
o = labelDict['label_family_history']
g = sns.factorplot(x="family_history", y="treatment", hue="Gender", data=train_df,
g.set_xticklabels(o)
plt.title('Probability of mental health condition')
plt.ylabel('Probability x 100')
plt.xlabel('Family History')

# 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()
```

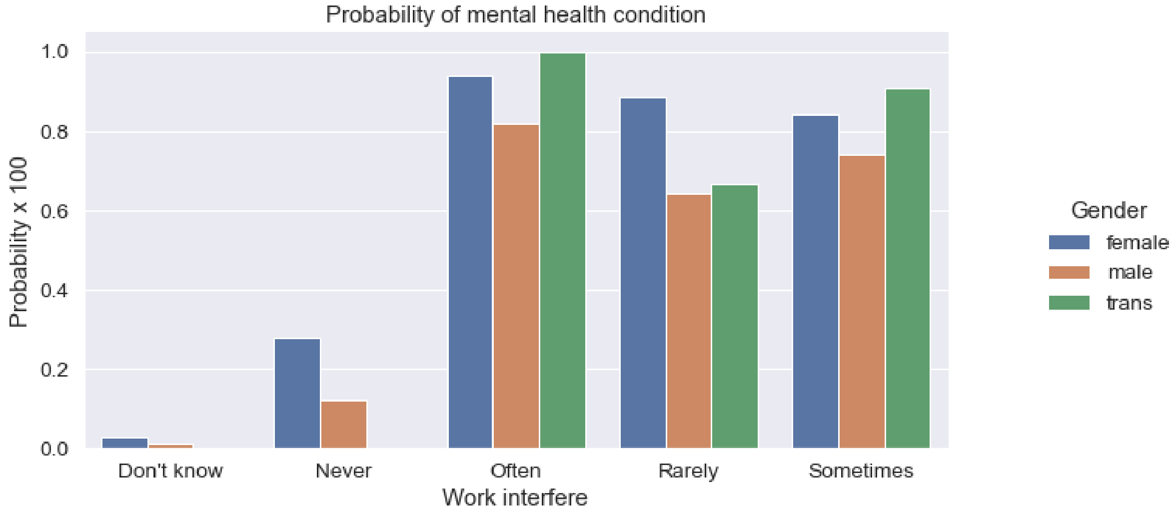
```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\categorical.py:3717: UserWarnin
g: The `factorplot` function has been renamed to `catplot`. The original name will
be removed in a future release. Please update your code. Note that the default `ki
nd` in `factorplot` (`'point'`) has changed `'strip'` in `catplot`.
  warnings.warn(msg)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\categorical.py:3723: UserWarnin
g: The `size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```



In [205…

```
#Barplot to show probabilities for work interfere

o = labelDict['label_work_interfere']
g = sns.factorplot(x="work_interfere", y="treatment", hue="Gender", data=train_df,
g.set_xticklabels(o)
plt.title('Probability of mental health condition')
plt.ylabel('Probability x 100')
plt.xlabel('Work interfere')

# 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()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\categorical.py:3717: UserWarnin
g: The `factorplot` function has been renamed to `catplot`. The original name will
be removed in a future release. Please update your code. Note that the default `ki
nd` in `factorplot` (`'point'`) has changed `'strip'` in `catplot`.
  warnings.warn(msg)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\categorical.py:3723: UserWarnin
g: The `size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```



In [206…

```
#Features Scaling We're going to scale age, because is extremely different from the

# Scaling Age
scaler = MinMaxScaler()
train_df['Age'] = scaler.fit_transform(train_df[['Age']])
train_df.head()
```

Out[206]:

	Age	Gender	self_employed	family_history	treatment	work_interfere	no_employees	rer
0	0.431818	0	0	0	1	2	4	
1	0.590909	1	0	0	0	3	5	
2	0.318182	1	0	0	0	3	4	
3	0.295455	1	0	1	1	2	2	
4	0.295455	1	0	0	0	1	1	

5 rows × 24 columns

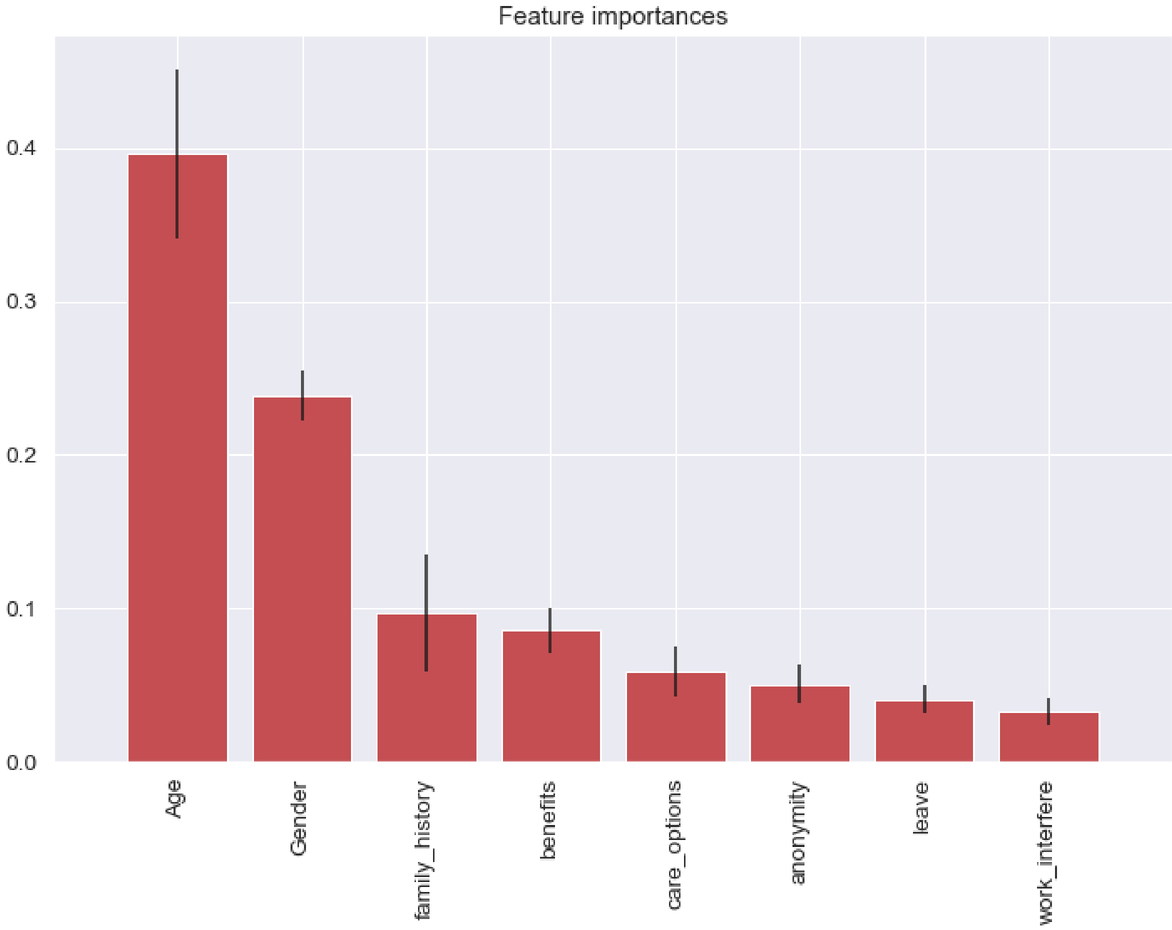
In [207…

```
# define X and y
feature_cols = ['Age', 'Gender', 'family_history', 'benefits', 'care_options', 'an
X = train_df[feature_cols]
y = train_df.treatment

# split X and y into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_s

# Create dictionaries for final graph
# Use: methodDict['Stacking'] = accuracy_score
methodDict = {}
rmseDict = ()
```

In [208…

```
# Build a forest and compute the feature importances
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])

# Plot the feature importances of the forest
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()
```



In [209…

```
#Tuning
def evalClassModel(model, y_test, y_pred_class, plot=False):
    #Classification accuracy: percentage of correct predictions
    # calculate accuracy
    print('Accuracy:', metrics.accuracy_score(y_test, y_pred_class))

    #Null accuracy: accuracy that could be achieved by always predicting the most f
    # examine the class distribution of the testing set (using a Pandas Series meth
    print('Null accuracy:\n', y_test.value_counts())

    # calculate the percentage of ones
    print('Percentage of ones:', y_test.mean())

    # calculate the percentage of zeros
    print('Percentage of zeros:',1 - y_test.mean())

    #Comparing the true and predicted response values
    print('True:', y_test.values[0:25])
    print('Pred:', y_pred_class[0:25])

    #Confusion matrix
    # save confusion matrix and slice into four pieces
```

```
    confusion = metrics.confusion_matrix(y_test, y_pred_class)
    #[row, column]
    TP = confusion[1, 1]
    TN = confusion[0, 0]
    FP = confusion[0, 1]
    FN = confusion[1, 0]
    # visualize Confusion Matrix
    sns.heatmap(confusion,annot=True,fmt="d")
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

    #Metrics computed from a confusion matrix
    #Classification Accuracy: Overall, how often is the classifier correct?
    accuracy = metrics.accuracy_score(y_test, y_pred_class)
    print('Classification Accuracy:', accuracy)

    #Classification Error: Overall, how often is the classifier incorrect?
    print('Classification Error:', 1 - metrics.accuracy_score(y_test, y_pred_class

    #False Positive Rate: When the actual value is negative, how often is the pred
    false_positive_rate = FP / float(TN + FP)
    print('False Positive Rate:', false_positive_rate)

    #Precision: When a positive value is predicted, how often is the prediction co
    print('Precision:', metrics.precision_score(y_test, y_pred_class))


    # IMPORTANT: first argument is true values, second argument is predicted proba
    print('AUC Score:', metrics.roc_auc_score(y_test, y_pred_class))

    # calculate cross-validated AUC
    print('Cross-validated AUC:', cross_val_score(model, X, y, cv=10, scoring='roc_

    ##########################################
    #Adjusting the classification threshold
    ##########################################
    # print the first 10 predicted responses
    print('First 10 predicted responses:\n', model.predict(X_test)[0:10])
        # print the first 10 predicted probabilities of class membership
    print('First 10 predicted probabilities of class members:\n', model.predict_pr

    # print the first 10 predicted probabilities for class 1
    model.predict_proba(X_test)[0:10, 1]

    # store the predicted probabilities for class 1
    y_pred_prob = model.predict_proba(X_test)[:, 1]

    if plot == True:
        # histogram of predicted probabilities
        plt.rcParams['font.size'] = 12
        plt.hist(y_pred_prob, bins=8)

        # x-axis limit from 0 to 1
        plt.xlim(0,1)
        plt.title('Histogram of predicted probabilities')
        plt.xlabel('Predicted probability of treatment')
        plt.ylabel('Frequency')


    # predict treatment if the predicted probability is greater than 0.3
    # it will return 1 for all values above 0.3 and 0 otherwise
    # results are 2D so we slice out the first column
```

```
    y_pred_prob = y_pred_prob.reshape(-1,1)
    y_pred_class = binarize(y_pred_prob)[0]

    # print the first 10 predicted probabilities
    print('First 10 predicted probabilities:\n', y_pred_prob[0:10])

    ##########################################
    #ROC Curves and Area Under the Curve (AUC)
    ##########################################

    #AUC is the percentage of the ROC plot that is underneath the curve
    #Higher value = better classifier
    roc_auc = metrics.roc_auc_score(y_test, y_pred_prob)



    # IMPORTANT: first argument is true values, second argument is predicted probal
    # roc_curve returns 3 objects fpr, tpr, thresholds
    # fpr: false positive rate
    # tpr: true positive rate
    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)' % r
        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()

    # define a function that accepts a threshold and prints sensitivity and specif
    def evaluate_threshold(threshold):
        #Sensitivity: When the actual value is positive, how often is the predictic
        #Specificity: When the actual value is negative, how often is the predictic
        print('Specificity for ' + str(threshold) + ' :', 1 - fpr[thresholds > thr

    # One way of setting threshold
    predict_mine = np.where(y_pred_prob > 0.50, 1, 0)
    confusion = metrics.confusion_matrix(y_test, predict_mine)
    print(confusion)



    return accuracy
```

In [210…

```
#Tuning with cross validation score

def tuningCV(knn):

    # search for an optimal value of K for 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)
    # plot the value of K for KNN (x-axis) versus the cross-validated accuracy (y-a
```

```
    plt.plot(k_range, k_scores)
    plt.xlabel('Value of K for KNN')
    plt.ylabel('Cross-Validated Accuracy')
    plt.show()
```

In [211…

```
#Tuning with GridSearchCV

def tuningGridSerach(knn):
    #More efficient parameter tuning using GridSearchCV
    k_range = list(range(1, 31))
    print(k_range)

    # create a parameter grid: map the parameter names to the values that should be
    param_grid = dict(n_neighbors=k_range)
    print(param_grid)

    # instantiate the grid
    grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')

    # fit the grid with data
    grid.fit(X, y)

    # view the complete results (list of named tuples)
    grid.grid_scores_

    # examine the first tuple
    print(grid.grid_scores_[0].parameters)
    print(grid.grid_scores_[0].cv_validation_scores)
    print(grid.grid_scores_[0].mean_validation_score)

    # create a list of the mean scores only
    grid_mean_scores = [result.mean_validation_score for result in grid.grid_scores
    print(grid_mean_scores)
    # plot the results
    plt.plot(k_range, grid_mean_scores)
    plt.xlabel('Value of K for KNN')
    plt.ylabel('Cross-Validated Accuracy')
    plt.show()

    # examine the best model
    print('GridSearch best score', grid.best_score_)
    print('GridSearch best params', grid.best_params_)
    print('GridSearch best estimator', grid.best_estimator_)
```

In [212…

```
#Tuning with RandomizedSearchCV

def tuningRandomizedSearchCV(model, param_dist):
    #Searching multiple parameters simultaneously
    # n_iter controls the number of searches
    rand = RandomizedSearchCV(model, param_dist, cv=10, scoring='accuracy', n_iter
    rand.fit(X, y)
    rand.cv_results_

    # examine the best model
    print('Rand. Best Score: ', rand.best_score_)
    print('Rand. Best Params: ', rand.best_params_)

    # run RandomizedSearchCV 20 times (with n_iter=10) and record the best score
    best_scores = []
    for _ in range(20):
        rand = RandomizedSearchCV(model, param_dist, cv=10, scoring='accuracy', n_
        rand.fit(X, y)
```

```
        best_scores.append(round(rand.best_score_, 3))
    print(best_scores)
```

In [213…

```
#Tuning with searching multiple parameters simultaneously

def tuningMultParam(knn):

    #Searching multiple parameters simultaneously
    # define the parameter values that should be searched
    k_range = list(range(1, 31))
    weight_options = ['uniform', 'distance']

    # create a parameter grid: map the parameter names to the values that should be
    param_grid = dict(n_neighbors=k_range, weights=weight_options)
    print(param_grid)

    # instantiate and fit the grid
    grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')
    grid.fit(X, y)

    # view the complete results
    print(grid.grid_scores_)

    # examine the best model
    print('Multiparam. Best Score: ', grid.best_score_)
    print('Multiparam. Best Params: ', grid.best_params_)
```

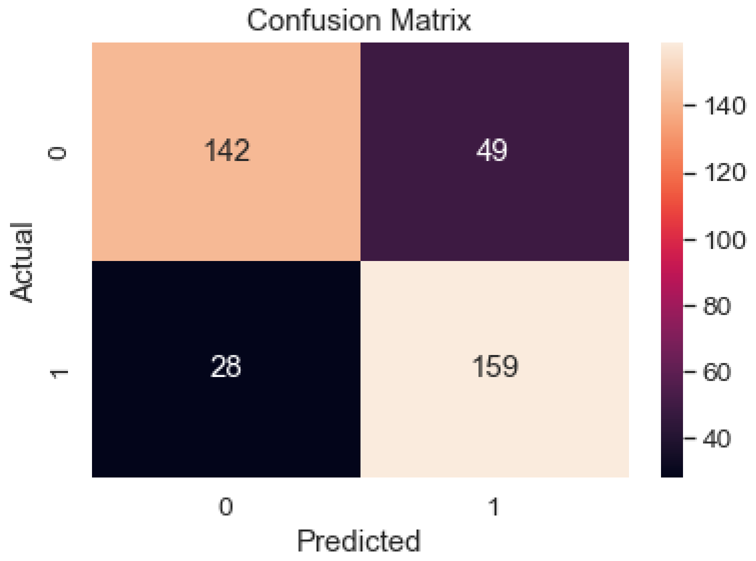
In [214…

```
#Evaluating models
#Logistic Regression

def logisticRegression():
    # train a logistic regression model on the training set
    logreg = LogisticRegression()
    logreg.fit(X_train, y_train)

    # make class predictions for the testing set
    y_pred_class = logreg.predict(X_test)

    accuracy_score = evalClassModel(logreg, y_test, y_pred_class, True)

    #Data for final graph
    methodDict['Log. Regression'] = accuracy_score * 100
```

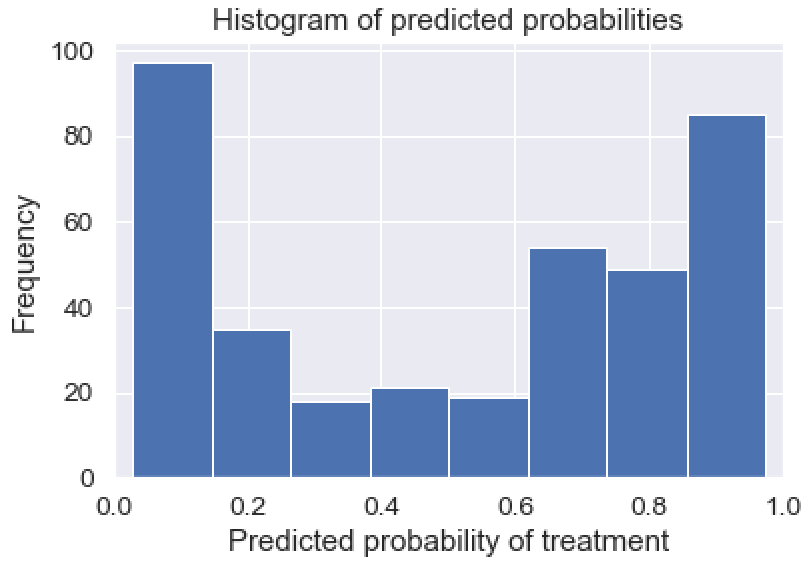
In [215…

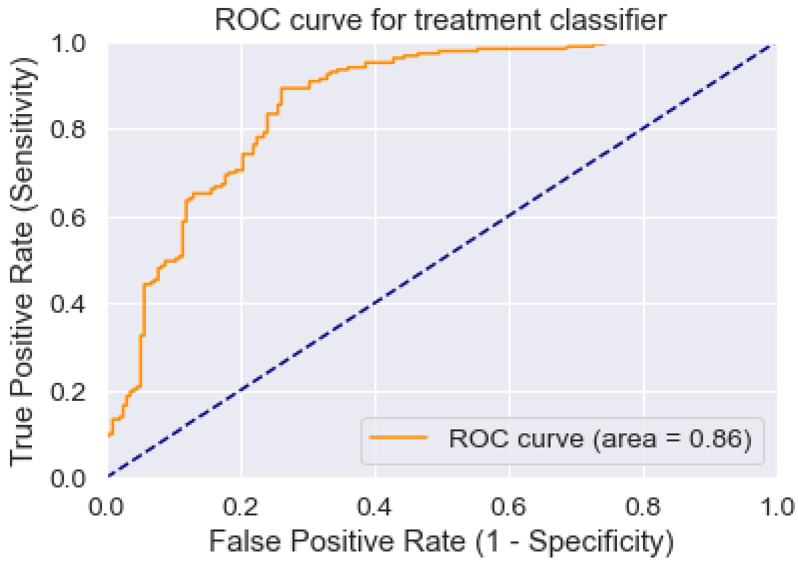
```
logisticRegression()
```

```
Accuracy: 0.7962962962962963
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 0 1 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0]
```




```
Classification Accuracy: 0.7962962962962963
Classification Error: 0.20370370370370372
False Positive Rate: 0.25654450261780104
Precision: 0.7644230769230769
AUC Score: 0.7968614385306716
Cross-validated AUC: 0.8753623882722146
First 10 predicted responses:
 [1 0 0 0 1 1 0 1 0 1]
First 10 predicted probabilities of class members:
 [[0.09193053 0.90806947]
 [0.95991564 0.04008436]
 [0.96547467 0.03452533]
 [0.78757121 0.21242879]
 [0.38959922 0.61040078]
 [0.05264207 0.94735793]
 [0.75035574 0.24964426]
 [0.19065116 0.80934884]
 [0.61612081 0.38387919]
 [0.47699963 0.52300037]]
First 10 predicted probabilities:
 [[0.90806947]
 [0.04008436]
 [0.03452533]
 [0.21242879]
 [0.61040078]
 [0.94735793]
 [0.24964426]
 [0.80934884]
 [0.38387919]
 [0.52300037]]
```






```
[[142  49]
 [ 28 159]]
```

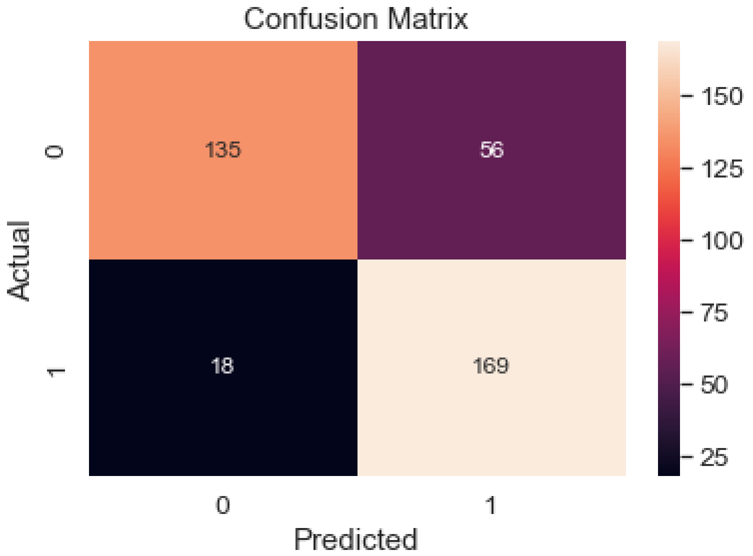
In [216…

```
#KNeighbors Classifier

def Knn():
    # Calculating the best parameters
    knn = KNeighborsClassifier(n_neighbors=5)

    # define the parameter values that should be searched
    k_range = list(range(1, 31))
    weight_options = ['uniform', 'distance']

    # specify "parameter distributions" rather than a "parameter grid"
    param_dist = dict(n_neighbors=k_range, weights=weight_options)
    tuningRandomizedSearchCV(knn, param_dist)

    # train a KNeighborsClassifier model on the training set
    knn = KNeighborsClassifier(n_neighbors=27, weights='uniform')
    knn.fit(X_train, y_train)

    # make class predictions for the testing set
    y_pred_class = knn.predict(X_test)

    accuracy_score = evalClassModel(knn, y_test, y_pred_class, True)

    #Data for final graph
    methodDict['K-Neighbors'] = accuracy_score * 100
```

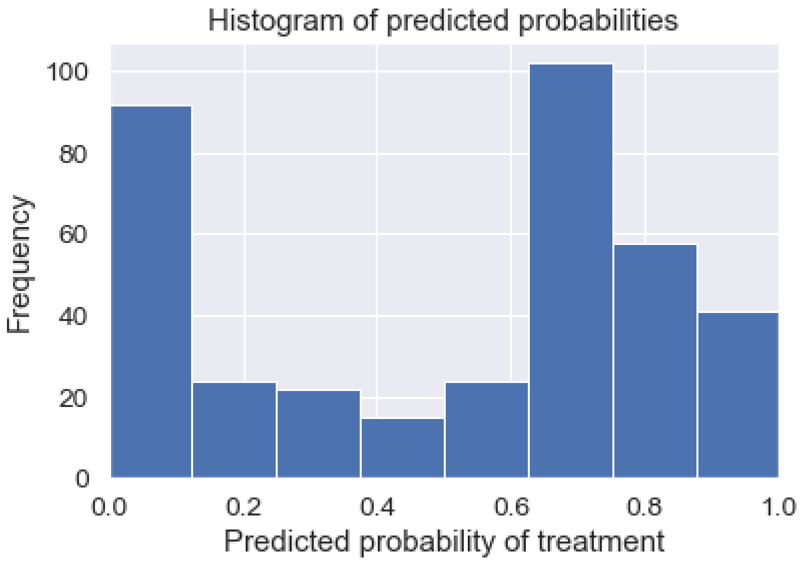
In [217…

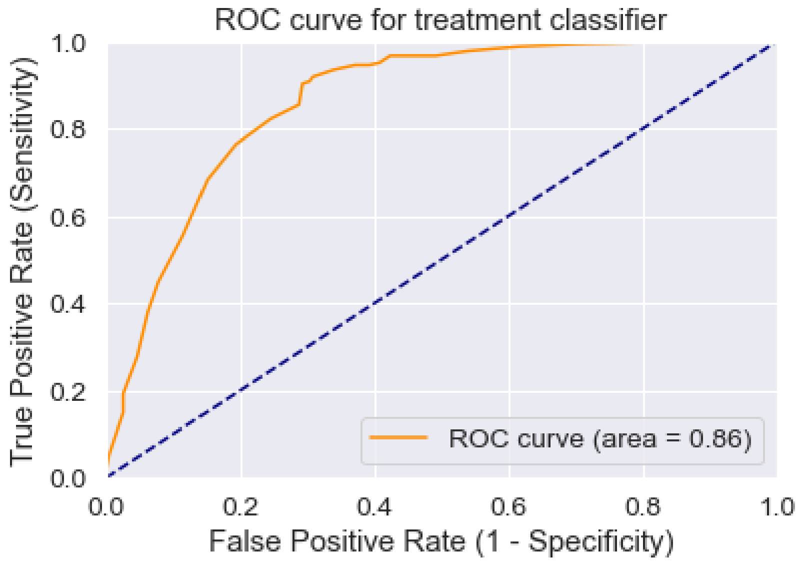
```
Knn()
```

```
Rand. Best Score:  0.8217650793650794
Rand. Best Params:  {'weights': 'uniform', 'n_neighbors': 27}
[0.819, 0.817, 0.819, 0.822, 0.822, 0.816, 0.814, 0.822, 0.815, 0.816, 0.822, 0.81
9, 0.819, 0.816, 0.822, 0.817, 0.819, 0.814, 0.822, 0.819]
Accuracy: 0.8042328042328042
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: 0.8042328042328042
Classification Error: 0.1957671957671958
False Positive Rate: 0.2931937172774869
Precision: 0.7511111111111111
AUC Score: 0.8052747991152673
Cross-validated AUC: 0.8784644661702792
First 10 predicted responses:
 [1 0 0 0 1 1 0 1 1 1]
First 10 predicted probabilities of class members:
 [[0.33333333 0.66666667]
 [1.         0.        ]
 [1.         0.        ]
 [0.66666667 0.33333333]
 [0.37037037 0.62962963]
 [0.03703704 0.96296296]
 [0.59259259 0.40740741]
 [0.37037037 0.62962963]
 [0.33333333 0.66666667]
 [0.33333333 0.66666667]]
First 10 predicted probabilities:
 [[0.66666667]
 [0.        ]
 [0.        ]
 [0.33333333]
 [0.62962963]
 [0.96296296]
 [0.40740741]
 [0.62962963]
 [0.66666667]
 [0.66666667]]
```






```
[[135  56]
 [ 18 169]]
```

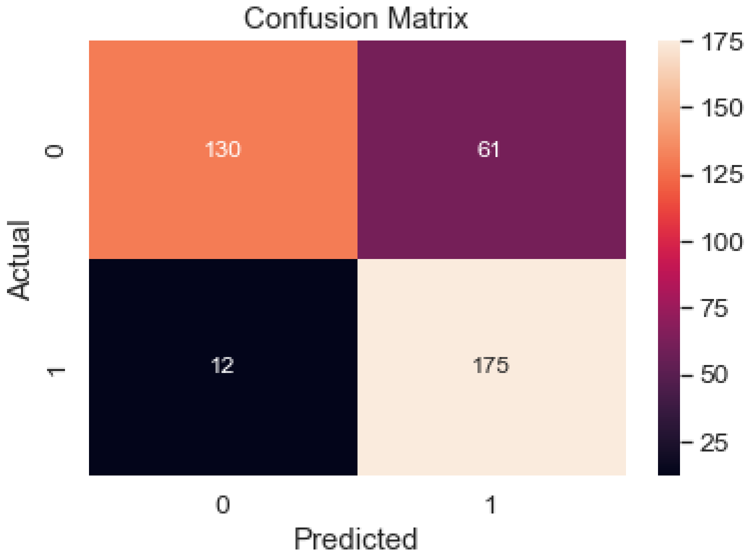
In [218…

```
def treeClassifier():
    # Calculating the best parameters
    tree = DecisionTreeClassifier()
    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(tree, param_dist)

    # train a decision tree model on the training set
    tree = DecisionTreeClassifier(max_depth=3, min_samples_split=8, max_features=6,
    tree.fit(X_train, y_train)

    # make class predictions for the testing set
    y_pred_class = tree.predict(X_test)

    accuracy_score = evalClassModel(tree, y_test, y_pred_class, True)

    #Data for final graph
    methodDict['Decision Tree Classifier'] = accuracy_score * 100
```

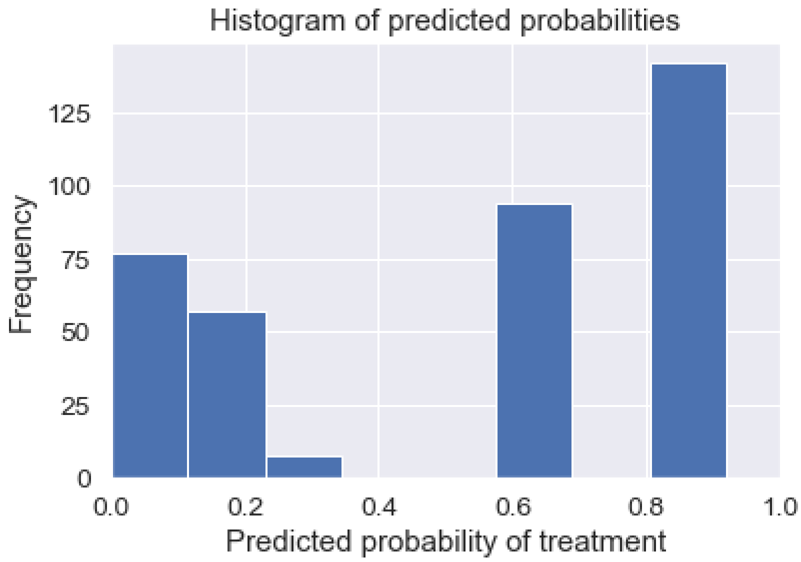
In [219…

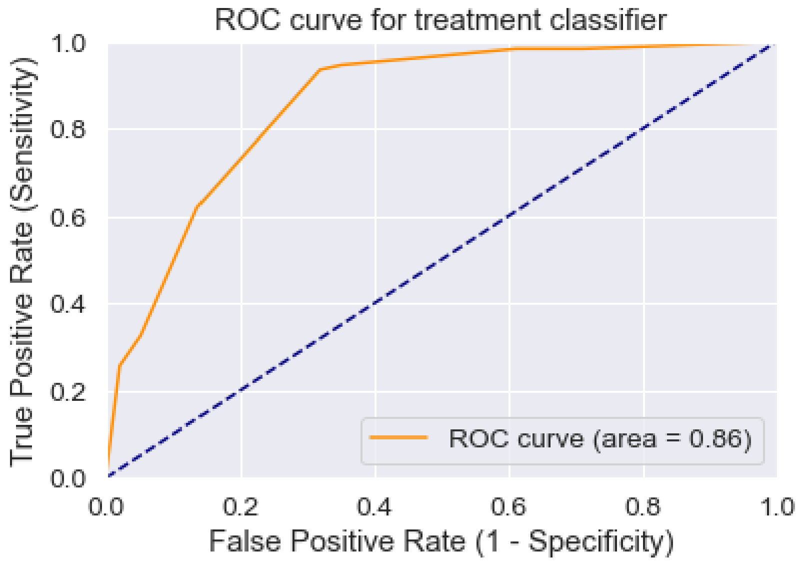
```
treeClassifier()
```

```
Rand. Best Score:  0.8305206349206349
Rand. Best Params:  {'criterion': 'entropy', 'max_depth': 3, 'max_features': 6, 'm
in_samples_leaf': 4, 'min_samples_split': 3}
[0.829, 0.831, 0.824, 0.817, 0.831, 0.829, 0.831, 0.831, 0.807, 0.831, 0.831, 0.83
1, 0.83, 0.826, 0.826, 0.831, 0.83, 0.831, 0.811, 0.831]
Accuracy: 0.8068783068783069
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: 0.8068783068783069
Classification Error: 0.19312169312169314
False Positive Rate: 0.3193717277486911
Precision: 0.7415254237288136
AUC Score: 0.8082285746283282
Cross-validated AUC: 0.8880490224390234
First 10 predicted responses:
 [1 0 0 0 1 1 0 1 1 1]
First 10 predicted probabilities of class members:
 [[0.18       0.82      ]
 [0.97959184 0.02040816]
 [1.         0.        ]
 [0.8778626  0.1221374 ]
 [0.36097561 0.63902439]
 [0.18       0.82      ]
 [0.8778626  0.1221374 ]
 [0.11320755 0.88679245]
 [0.36097561 0.63902439]
 [0.36097561 0.63902439]]
First 10 predicted probabilities:
 [[0.82      ]
 [0.02040816]
 [0.        ]
 [0.1221374 ]
 [0.63902439]
 [0.82      ]
 [0.1221374 ]
 [0.88679245]
 [0.63902439]
 [0.63902439]]
```





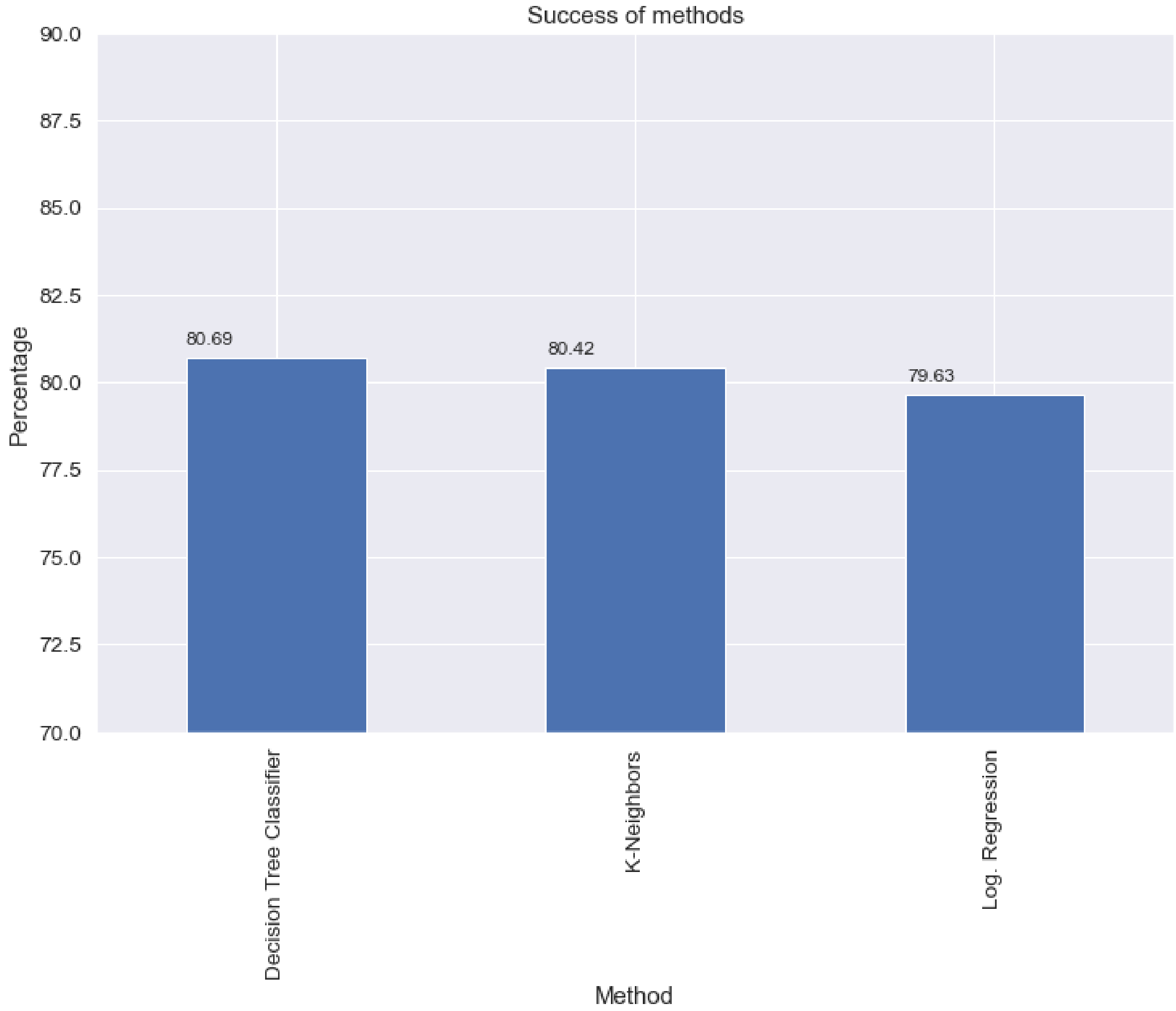

```
[[130  61]
 [ 12 175]]
```

In [220…

```
def plotSuccess():
    s = pd.Series(methodDict)
    s = s.sort_values(ascending=False)
    plt.figure(figsize=(12,8))
    #Colors
    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
    plt.ylim([70.0, 90.0])
    plt.xlabel('Method')
    plt.ylabel('Percentage')
    plt.title('Success of methods')

    plt.show()
```

In [221…

```
plotSuccess()
```



In [222…

```
# Generate predictions with the best method
clf = AdaBoostClassifier()
clf.fit(X, y)
dfTestPredictions = clf.predict(X_test)

# Write predictions to csv file
# We don't have any significative field so we save the index
results = pd.DataFrame({'Index': X_test.index, 'Treatment': dfTestPredictions})
# Save to file
# This file will be visible after publishing in the output section
results.to_csv('results.csv', index=False)
results.head(50)
```

Out[222]:

	Index	Treatment
0	5	1
1	494	0
2	52	0
3	984	0
4	186	0
5	18	1
6	317	0
7	511	1
8	364	1
9	571	1
10	609	0
11	1147	1
12	922	1
13	461	0
14	740	1
15	955	1
16	814	1
17	1160	1
18	85	0
19	733	0
20	1112	0
21	124	0
22	1040	1
23	492	0
24	1159	0
25	211	1
26	1020	0
27	892	0
28	453	0
29	646	1
30	161	1
31	811	1
32	1104	0
33	45	1
34	1241	1
35	874	0

	Index	Treatment
36	921	1
37	1191	1
38	481	1
39	308	0
40	269	0
41	731	0
42	1017	1
43	1193	0
44	875	1
45	1	1
46	796	1
47	1092	1
48	141	1
49	1231	0

In [ ]:

```

```