Descrizione del problema

In questo progetto si vuole realizzare un modello in grado di predire la potabilità dell'acqua in base a delle analisi effettuate in laboratorio

Importo le librerie necessarie

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os.path
from urllib.request import urlretrieve
import random
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import mean_squared_error
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
```

Caricamento dei dati

```
dataset_url = "https://github.com/aleshark87/data-intensive-project/raw/main/water
dataset_filename = "water_potability.csv"

if not os.path.exists(dataset_filename):
  urlretrieve(dataset_url, dataset_filename)

dataset = pd.read_csv(dataset_filename, sep=",")
dataset.head(10)
```

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Orga
0	NaN	204.890455	20791.318981	7.300212	368.516441	564.308654	

Descrizione delle feature

Unità di misura:

- **NTU**: Nephelometric Turbidity unit
- **ppm**: Parti per milione
- **µg/L**: Microgrammi per litro
- **mg/L**: Milligrammi per litro

Descrizione:

1. **ph**: Misura del pH dell'acqua.
2. **Hardness**: Misura della durezza dell'acqua. Viene espressa in mg/L.
3. **Solids**: Misura dei materiali disciolti. Viene espressa in ppm.
4. **Chloramines**: Misura delle clorammine in ppm.
5. **Sulfate**: Misura dei solfati in mg/L.
6. **Conductivity**: Conduttività elettrica dell'acqua in µS/cm.
7. **Organic_carbon**: Carbonio organico in ppm.
8. **Trihalomethanes**: Misura dei trialometani in µg/L.
9. **Turbidity**: Misura della torbidità in NTU.
10. **Potability**: Indica se l'acqua è potabile, 1 significa potabile e 0 significa non potabile.

Analisi generale dei dati

```
dataset.describe()
```

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivi
count	2785.000000	3276.000000	3276.000000	3276.000000	2495.000000	3276.0000
mean	7.080795	196.369496	22014.092526	7.122277	333.775777	426.2051
std	1.594320	32.879761	8768.570828	1.583085	41.416840	80.8240
min	0.000000	47.432000	320.942611	0.352000	129.000000	181.4837
25%	6.093092	176.850538	15666.690297	6.127421	307.699498	365.7344
50%	7.036752	196.967627	20927.833607	7.130299	333.073546	421.8849
75%	8.062066	216.667456	27332.762127	8.114887	359.950170	481.7923
max	14.000000	323.124000	61227.196008	13.127000	481.030642	753.3426

Possiamo notare che i dati in nostro possesso sono tutti numerici, e di diversa scala uno dall'altro. Sarà quindi fondamentale procedere allo scaling prima di costruire i modelli. Inoltre, si nota la presenza di valori nulli e di sbilanciamento nella classe **Potability**.

Vediamo quanti sono i valori nulli all'interno del dataset.

```
dataset.isnull().sum()
```

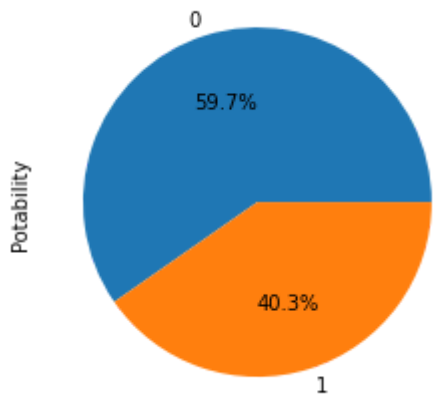
```
ph                 491
Hardness             0
Solids               0
Chloramines          0
Sulfate            781
Conductivity         0
Organic_carbon       0
Trihalomethanes    162
Turbidity            0
Potability           0
dtype: int64
```

Procediamo alla rimozione.

```
dataset.dropna(inplace=True)
dataset.isnull().sum()
```

```
ph                 0
Hardness           0
Solids             0
Chloramines        0
Sulfate            0
Conductivity       0
Organic_carbon     0
Trihalomethanes    0
Turbidity          0
Potability         0
dtype: int64
```

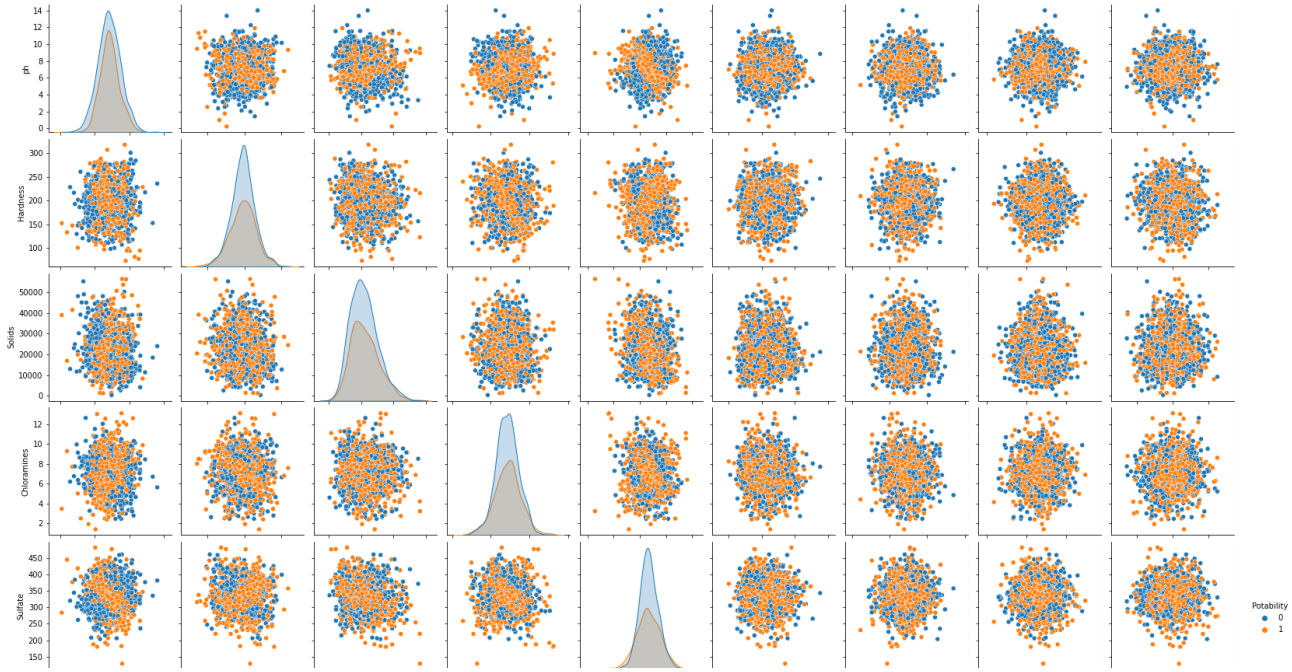
```
dataset.Potability.value_counts().plot.pie(autopct="%.1f%%");
```



Da questo grafico a torta possiamo vedere meglio la differenza di distribuzione nelle classi.

```
sns.pairplot(dataset, hue="Potability")
```

```
<seaborn.axisgrid.PairGrid at 0x7fcd3d90c550>
```



Dal pairplot possiamo verificare:

> La distribuzione delle feature a nostra disposizione, viste singolarmente, in funzione della potabilità dell'acqua.

> La distribuzione delle feature comparate tra loro, in ogni combinazione possibile. Anche in questo caso il colore del dato sul grafico rappresenta la potabilità dell'acqua.

La distribuzione delle feature è omogenea, come sembra anche la correlazione tra le feature(i dati incrociati sono omogenei nelle distribuzione).

```
corr = dataset.corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
sns.heatmap(corr, mask=mask, cmap="Reds");
```

Anche la heatmap delle correlazioni, che dovrebbe mostrarci correlazioni tra le feature, non ne mostra alcuna rilevante. Modelli con feature non lineari dovrebbero performare meglio.

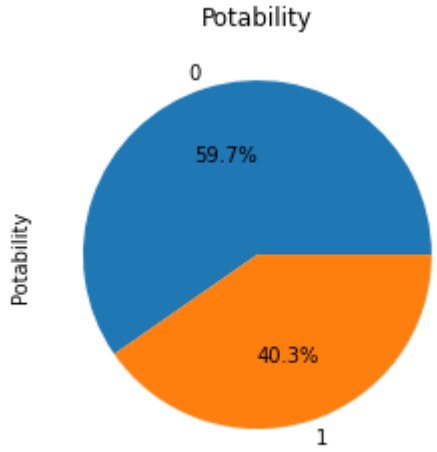
Bilanciamento classi e Regolarizzazione

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

```
X = dataset.drop(columns="Potability")
y = dataset["Potability"]
```

```
pd.value_counts(y).plot.pie(autopct="%.1f%%", title="Potability")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fcd3b7bc690>
```

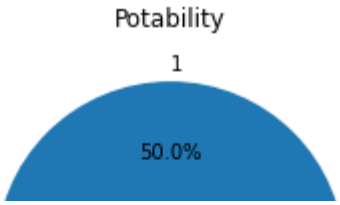



```
sm = SMOTE(random_state=42)
X_balanced, y_balanced = sm.fit_resample(X, y)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: Futur
  warnings.warn(msg, category=FutureWarning)
```

```
pd.value_counts(y_balanced).plot.pie(autopct="%.1f%%", title="Potability")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fcd3b7bac50>
```



Procediamo dunque suddividendo i dati in training e validation set tramite metodo holdout:

```
X_train, X_val, y_train, y_val = train_test_split(
    X_balanced,y_balanced,
    test_size = 1/3,
    random_state = 42
)
```

Perceptron non standardizzato

```
not_std_perc = Pipeline([
                  ("perc", Perceptron(random_state=42))
])
```

```
not_std_perc.fit(X_train, y_train)
print("R-squared coefficient:")
not_std_perc.score(X_val, y_val)
```

```
R-squared coefficient:
0.4775
```

Perceptron standardizzato

```
std_perc = Pipeline([
  ("scaler", StandardScaler()),
    ("perc", Perceptron(random_state=42))
])
```

```
std_perc.fit(X_train, y_train)
print("R-squared coefficient:")
std_perc.score(X_val, y_val)
```

```
R-squared coefficient:
0.5075
```

Standardizzando le feature il tasso di accuratezza sale, infatti avevamo già visto in precedenza che avevano ordini di grandezza molto differenti tra loro. Ora proviamo la **regolarizzazione L1**.

```
std_l1_model = Pipeline([
                      ("scaler", StandardScaler()),
                      ("perc", Perceptron(penalty="l1", n_jobs=-1, random_state=42
```

```
])
```

```
std_l1_model.fit(X_train, y_train)
print("R-squared coefficient:")
std_l1_model.score(X_val, y_val)
```

```
R-squared coefficient:
0.4775
```

La regolarizzazione abbassa il tasso di accuratezza, quindi ho deciso di non eliminare feature dal dataset.

```
col_coef = pd.DataFrame(std_l1_model.named_steps["perc"].coef_[0], columns=["coeff
col_coef
```

	coefficients
ph	0.0
Hardness	0.0
Solids	0.0
Chloramines	0.0
Sulfate	0.0
Conductivity	0.0
Organic_carbon	0.0
Trihalomethanes	0.0
Turbidity	0.0

Considerando questi coefficienti e la non correlazione delle feature, effettuo l'analisi delle feature più significative attraverso un modello non lineare.

```
from xgboost import XGBClassifier

xgb_model = Pipeline([
    ("scaler", StandardScaler()),
    ("xgb", XGBClassifier(nthread=8, objective='binary:logistic', random_state=42)
])

xgb_model.fit(X_train, y_train)
```

```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('xgb',
                 XGBClassifier(base_score=0.5, booster='gbtree',
                               colsample_bylevel=1, colsample_bynode=1,
```

```
                                colsample_bytree=1, gamma=0, learning_rate=0.1
                                max_delta_step=0, max_depth=3,
                                min_child_weight=1, missing=None,
                                n_estimators=100, n_jobs=1, nthread=8,
                                objective='binary:logistic', random_state=42,
                                reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
                                seed=None, silent=None, subsample=1,
                                verbosity=1))],
                    verbose=False)
```

```
xgb_model.score(X_val, y_val)
```

```
0.655
```

Vediamo le feature più significative.

```
col_coef = pd.DataFrame(xgb_model["xgb"].feature_importances_, columns=["coefficie
col_coef
```

	coefficients
ph	0.161084
Hardness	0.090977
Solids	0.134421
Chloramines	0.140948
Sulfate	0.155884
Conductivity	0.078240
Organic_carbon	0.075096
Trihalomethanes	0.082683
Turbidity	0.080666

Nessuna feature ha importanza 0, quindi ho deciso di non eliminarne nessuna.

Modellazione

```
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
```

Creiamo dei dizionari per memorizzare informazioni sui modelli

```
scores = {}
f1_scores = {}
precision = {}
recall = {}
models = []
confusion_m = {}
mse = {}
```

Perceptron

```
models.append("Perceptron")

model = Pipeline([
    ("scaler", StandardScaler()),
    ("perc", Perceptron(random_state=42))
])

param = {
    "scaler": [StandardScaler()],
    "perc__penalty": [None, "l2", "l1"],
    "perc__fit_intercept": [False, True]
}


skf = StratifiedKFold(3, shuffle=True, random_state=42)


perc_gv = GridSearchCV(model, param, cv=skf)
perc_gv.fit(X_train, y_train)
```

```
GridSearchCV(cv=StratifiedKFold(n_splits=3, random_state=42, shuffle=True),
             error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('scaler',
                                        StandardScaler(copy=True,
                                                       with_mean=True,
                                                       with_std=True)),
                                       ('perc',
                                        Perceptron(alpha=0.0001,
                                                   class_weight=None,
                                                   early_stopping=False,
                                                   eta0=1.0, fit_intercept=Tr
                                                   max_iter=1000,
                                                   n_iter_no_change=5,
                                                   n_jobs=None, penalty=None,
                                                   shuffle=True, tol=0.001,
                                                   validation_fraction=0.1,
                                                   verbose=0,
                                                   warm_start=False))],
                                verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'perc__fit_intercept': [False, True],
                         'perc__penalty': [None, 'l2', 'l1'],
                         'scaler': [StandardScaler(copy=True, with_mean=True,
```

```
                                                    with_std=True)]},
                 pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                 scoring=None, verbose=0)
```

```
print("Punteggio migliore: {score}".format(score=perc_gv.score(X_val, y_val)));
print("F1 score: {score}".format(score=f1_score(y_val, perc_gv.predict(X_val), ave
print("Precision score: {score}".format(score=precision_score(y_val, perc_gv.predi
print("Recall score: {score}".format(score=recall_score(y_val, perc_gv.predict(X_v
print("Parametri migliori: {params}".format(params=perc_gv.best_params_))

modelName = models[len(models)-1]
scores[modelName] = perc_gv.score(X_val, y_val);
f1_scores[modelName] = f1_score(y_val, perc_gv.predict(X_val), average="binary")
precision[modelName] = precision_score(y_val, perc_gv.predict(X_val))
recall[modelName] = recall_score(y_val, perc_gv.predict(X_val))
```

```
Punteggio migliore: 0.50125
F1 score: 0.5221556886227544
Precision score: 0.5227817745803357
Recall score: 0.5215311004784688
Parametri migliori: {'perc__fit_intercept': False, 'perc__penalty': 'l1', 'sc
```

```
index = ["Not potable", "Potable"]
classes = ["Not potable[P]", "Potable[P]"]
confusion_m[modelName] = pd.DataFrame(confusion_matrix(y_val, perc_gv.predict(X_va
print(confusion_m[modelName])
mse[modelName] = mean_squared_error(y_val, perc_gv.predict(X_val))
```

```
                Not potable[P]  Potable[P]
Not potable               183         199
Potable                   200         218
```

Regressione Logistica

```
models.append("Logistic Regression")

log_model = Pipeline([
    ("scaler", StandardScaler()),
    ("log_reg", LogisticRegression(solver='saga', random_state=42))
])

log_grid = {
    "scaler": [StandardScaler()],
    "log_reg__penalty": ["l2", "l1"],
    "log_reg__C": np.logspace(-3, 3, 10),
    "log_reg__fit_intercept": [False, True]
}


skf = StratifiedKFold(3, shuffle=True, random_state=42)
log_gv = GridSearchCV(log_model, log_grid, cv=skf, n_jobs=-1)
log_gv.fit(X_train, y_train)
```

```
GridSearchCV(cv=StratifiedKFold(n_splits=3, random_state=42, shuffle=True),
             error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('scaler',
                                        StandardScaler(copy=True,
                                                       with_mean=True,
                                                       with_std=True)),
                                       ('log_reg',
                                        LogisticRegression(C=1.0,
                                                           class_weight=None,
                                                           dual=False,
                                                           fit_intercept=True
                                                           intercept_scaling=
                                                           l1_ratio=None,
                                                           max_iter=100,
                                                           multi_class='auto'
                                                           n_jobs=...
             param_grid={'log_reg__C': array([1.00000000e-03, 4.64158883e-03,
       4.64158883e-01, 2.15443469e+00, 1.00000000e+01, 4.64158883e+01,
       2.15443469e+02, 1.00000000e+03]),
                         'log_reg__fit_intercept': [False, True],
                         'log_reg__penalty': ['l2', 'l1'],
                         'scaler': [StandardScaler(copy=True, with_mean=True,
                                                   with_std=True)]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

```
print("Punteggio migliore: {score}".format(score=log_gv.score(X_val, y_val)));
print("F1 score: {score}".format(score=f1_score(y_val, log_gv.predict(X_val), aver
print("Precision score: {score}".format(score=precision_score(y_val, log_gv.predic
print("Recall score: {score}".format(score=recall_score(y_val, log_gv.predict(X_va
print("Parametri migliori: {params}".format(params=log_gv.best_params_))

modelName = models[len(models)-1]
scores[modelName] = log_gv.score(X_val, y_val);
f1_scores[modelName] = f1_score(y_val, log_gv.predict(X_val), average="binary")
precision[modelName] = precision_score(y_val, log_gv.predict(X_val))
recall[modelName] = recall_score(y_val, log_gv.predict(X_val))
```

```
Punteggio migliore: 0.51125
F1 score: 0.46947082767978293
Precision score: 0.542319749216301
Recall score: 0.4138755980861244
Parametri migliori: {'log_reg__C': 0.1, 'log_reg__fit_intercept': True, 'log_
```

```
index = ["Not potable", "Potable"]
classes = ["Not potable[P]", "Potable[P]"]
confusion_m[modelName] = pd.DataFrame(confusion_matrix(y_val, log_gv.predict(X_val
print(confusion_m[modelName])
mse[modelName] = mean_squared_error(y_val, log_gv.predict(X_val))
```

```
              Not potable[P]  Potable[P]
Not potable             236         146
Potable                 245         173
```

SVC

```
models.append("Support Vector Machines")

svc_model = Pipeline([
    ("scaler", StandardScaler()),
    ("svc", SVC(random_state=42))
])

param_svc = {
    'scaler': [StandardScaler()],
    'svc__kernel': ['linear'],
    'svc__kernel': ['rbf'],
    "svc__C": np.linspace(0.5, 5, 10)
}


skf = StratifiedKFold(3, shuffle=True, random_state=42)
svc_gv = GridSearchCV(svc_model, param_svc, cv=skf)
svc_gv.fit(X_train, y_train)
```

```
GridSearchCV(cv=StratifiedKFold(n_splits=3, random_state=42, shuffle=True),
             error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('scaler',
                                        StandardScaler(copy=True,
                                                       with_mean=True,
                                                       with_std=True)),
                                       ('svc',
                                        SVC(C=1.0, break_ties=False,
                                            cache_size=200, class_weight=None
                                            coef0=0.0,
                                            decision_function_shape='ovr',
                                            degree=3, gamma='scale',
                                            kernel='rbf', max_iter=-1,
                                            pr...lity=False, random_state=42,
                                            shrinking=True, tol=0.001,
                                            verbose=False))],
                                verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'scaler': [StandardScaler(copy=True, with_mean=True,
                                                   with_std=True)],
                         'svc__C': array([0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5,
                         'svc__kernel': ['rbf']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

```
print("Punteggio migliore: {score}".format(score=svc_gv.score(X_val, y_val)));
print("F1 score: {score}".format(score=f1_score(y_val, svc_gv.predict(X_val), aver
print("Precision score: {score}".format(score=precision_score(y_val, svc_gv.predic
print("Recall score: {score}".format(score=recall_score(y_val, svc_gv.predict(X_va
print("Parametri migliori: {params}".format(params=svc_gv.best_params_))
```

```
modelName = models[len(models)-1]
scores[modelName] = svc_gv.score(X_val, y_val);
f1_scores[modelName] = f1_score(y_val, svc_gv.predict(X_val), average="binary")
precision[modelName] = precision_score(y_val, svc_gv.predict(X_val))
recall[modelName] = recall_score(y_val, svc_gv.predict(X_val))
```

```
Punteggio migliore: 0.66625
F1 score: 0.6771463119709795
Precision score: 0.684596577017115
Recall score: 0.6698564593301436
Parametri migliori: {'scaler': StandardScaler(copy=True, with_mean=True, with
```

```
index = ["Not potable", "Potable"]
classes = ["Not potable[P]", "Potable[P]"]
confusion_m[modelName] = pd.DataFrame(confusion_matrix(y_val, svc_gv.predict(X_val
print(confusion_m[modelName])
mse[modelName] = mean_squared_error(y_val, svc_gv.predict(X_val))
```

```
              Not potable[P]  Potable[P]
Not potable             253         129
Potable                 138         280
```

Decision tree

```
models.append("Decision Tree")
num_features = X.columns.size

tree_model = Pipeline([
    ("scaler", StandardScaler()),
    ("tree", DecisionTreeClassifier(random_state=42))
])

tree_grid = {'scaler': [StandardScaler()],
             'tree__criterion': ['gini', 'entropy'],
             'tree__max_features': range(5, num_features)}


skf = StratifiedKFold(3, shuffle=True, random_state=42)
tree_gv = GridSearchCV(tree_model, tree_grid, cv=skf, n_jobs=-1)
tree_gv.fit(X_train, y_train)
```

```
GridSearchCV(cv=StratifiedKFold(n_splits=3, random_state=42, shuffle=True),
             error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('scaler',
                                        StandardScaler(copy=True,
                                                       with_mean=True,
                                                       with_std=True)),
                                       ('tree',
                                        DecisionTreeClassifier(ccp_alpha=0.0,
                                                               class_weight=N
                                                               criterion='gin
                                                               max_depth=None
                                                               max_features=N
```

```
                                                                max_leaf_nodes
                                                                min_impurity_d
                                                                min_weight_fra
                                                                presort='depre
                                                                random_state=4
                                                                splitter='best
                                verbose=False),
                 iid='deprecated', n_jobs=-1,
                 param_grid={'scaler': [StandardScaler(copy=True, with_mean=True,
                                                       with_std=True)],
                             'tree__criterion': ['gini', 'entropy'],
                             'tree__max_features': range(5, 9)},
                 pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                 scoring=None, verbose=0)
```

```
print("Punteggio migliore: {score}".format(score=tree_gv.score(X_val, y_val)));
print("F1 score: {score}".format(score=f1_score(y_val, tree_gv.predict(X_val), ave
print("Precision score: {score}".format(score=precision_score(y_val, tree_gv.predi
print("Recall score: {score}".format(score=recall_score(y_val, tree_gv.predict(X_v
print("Parametri migliori: {params}".format(params=tree_gv.best_params_))

modelName = models[len(models)-1]
scores[modelName] = tree_gv.score(X_val, y_val);
f1_scores[modelName] = f1_score(y_val, tree_gv.predict(X_val), average="binary")
precision[modelName] = precision_score(y_val, tree_gv.predict(X_val))
recall[modelName] = recall_score(y_val, tree_gv.predict(X_val))
```

```
Punteggio migliore: 0.60375
F1 score: 0.6194477791116447
Precision score: 0.6216867469879518
Recall score: 0.6172248803827751
Parametri migliori: {'scaler': StandardScaler(copy=True, with_mean=True, with
```

```
confusion_m[modelName] = pd.DataFrame(confusion_matrix(y_val, tree_gv.predict(X_va
print(confusion_m[modelName])
mse[modelName] = mean_squared_error(y_val, tree_gv.predict(X_val))
```

```
                Not potable[P]  Potable[P]
Not potable               225         157
Potable                   160         258
```

Xg Boost

```
models.append("XG Boost")

xgb_model = Pipeline([
    ("scaler", StandardScaler()),
    ("xgb", XGBClassifier(nthread=8, objective='binary:logistic', random_state=42)
])

parameters = {
    'xgb__max_depth': [13, 14],
    'xgb__grow_policy': ["lossguide", "depthwise"]
```

```
    'xgb__grow_policy' : ['lossguide', 'depthwise']
}

skf = StratifiedKFold(3, shuffle=True, random_state=42)
gs = GridSearchCV(xgb_model, parameters, cv=skf)

gs.fit(X_train, y_train)
y_pred = gs.predict(X_val)


print("Punteggio migliore: {score}".format(score=gs.score(X_val, y_val)));
print("Precision score: {score}".format(score=precision_score(y_val, y_pred)))
print("Recall score: {score}".format(score=recall_score(y_val, y_pred)))
print("Parametri migliori: {params}".format(params=gs.best_params_))

modelName = models[len(models)-1]
scores[modelName] = gs.score(X_val, y_val)
f1_scores[modelName] = f1_score(y_val, y_pred, average="binary")
precision[modelName] = precision_score(y_val, y_pred)
recall[modelName] = recall_score(y_val, y_pred)
```

```
Punteggio migliore: 0.6875
Precision score: 0.71875
Recall score: 0.6602870813397129
Parametri migliori: {'xgb__grow_policy': 'lossguide', 'xgb__max_depth': 13}
```

```
confusion_m[modelName] = pd.DataFrame(confusion_matrix(y_val, y_pred), index=index
print(confusion_m[modelName])
mse[modelName] = mean_squared_error(y_val, y_pred)
```

```
                Not potable[P]  Potable[P]
Not potable               274         108
Potable                   142         276
```

Valutazione dei modelli utilizzati

```
pd.DataFrame.from_dict(scores, orient="index", columns=["R^2 score"])
```

	R^2 score
Perceptron	0.50125
Logistic Regression	0.51125
Support Vector Machines	0.66625
Decision Tree	0.60375
XG Boost	0.68750

```
pd.DataFrame.from_dict(f1_scores, orient="index", columns=["F1 score"])
```

	F1 score
Perceptron	0.522156
Logistic Regression	0.469471
Support Vector Machines	0.677146
Decision Tree	0.619448

```
pd.DataFrame.from_dict(recall, orient="index", columns=["Recall score"])
```

	Recall score
Perceptron	0.521531
Logistic Regression	0.413876
Support Vector Machines	0.669856
Decision Tree	0.617225
XG Boost	0.660287

```
pd.DataFrame.from_dict(precision, orient="index", columns=["Precision score"])
```

	Precision score
Perceptron	0.522782
Logistic Regression	0.542320
Support Vector Machines	0.684597
Decision Tree	0.621687
XG Boost	0.718750

```
for i in range(len(models)):
  print(models[i])
  print(confusion_m[models[i]])
  print('\n')
```

```
    Perceptron
                 Not potable[P]  Potable[P]
    Not potable              183         199
    Potable                  200         218


    Logistic Regression
                 Not potable[P]  Potable[P]
    Not potable              236         146
    Potable                  245         173


    Support Vector Machines
                 Not potable[P]  Potable[P]
    Not potable              253         129
```

```
Potable                 138          280


Decision Tree
              Not potable[P]  Potable[P]
Not potable             225          157
Potable                 160          258


XG Boost
              Not potable[P]  Potable[P]
Not potable             274          108
Potable                 142          276
```

Confronto modelli con intervallo di confidenza

Quanto sono affidabili i modelli? Calcoliamo il valore di accuratezza in un intervallo di confidenza al 95%

```
def accuracy(confusion_m):
  return np.diag(confusion_m).sum() / confusion_m.sum().sum()


def confidence_interval(accuracy, N, Z):
  den = (2*(N+Z**2))
  variance = (Z*np.sqrt(Z**2+4*N*accuracy-4*N*accuracy**2)) / den
  a = (2*N*accuracy+Z**2) / den
  inf = a - variance
  sup = a + variance
  return np.round((inf, sup), 4)


def confidence_interval_between_models99(mse1, mse2):
  d = np.abs(mse1 - mse2)
  variance = (mse1 * (1 - mse1)) / len(X_val) + (mse2 * (1 - mse2)) / len(X_val)
  d_min = d - 2.58 * np.sqrt(variance)
  d_max = d + 2.58 * np.sqrt(variance)
  return np.round((d_min, d_max), 4)


#Intervallo di confidenza al 95%
Z = 1.96

pd.DataFrame([confidence_interval(accuracy(confusion_m[models[i]]), len(X_val), Z)
              index=models,
              columns=["inf", "sup"])
```

	inf	sup
Perceptron	0.4667	0.5358
Logistic Regression	0.4766	0.5458
Support Vector Machines	0.6328	0.6981

```
for i in range(len(models)):
  for k in range(i+1, len(models)):
    print(models[i] + " vs " + models[k])
    print(confidence_interval_between_models99(mse[models[i]], mse[models[k]]))
```

```
Perceptron vs Logistic Regression
[-0.0545  0.0745]
Perceptron vs Support Vector Machines
[0.1023 0.2277]
Perceptron vs Decision Tree
[0.0387 0.1663]
Perceptron vs XG Boost
[0.1241 0.2484]
Logistic Regression vs Support Vector Machines
[0.0923 0.2177]
Logistic Regression vs Decision Tree
[0.0287 0.1563]
Logistic Regression vs XG Boost
[0.1141 0.2384]
Support Vector Machines vs Decision Tree
[0.0005 0.1245]
Support Vector Machines vs XG Boost
[-0.0391  0.0816]
Decision Tree vs XG Boost
[0.0223 0.1452]
```

I modelli che performano meglio non presentano differenze significative.

Confronto con un modello casuale

```
from sklearn.dummy import DummyClassifier
```

```
random = DummyClassifier(strategy="uniform", random_state=42)
random.fit(X_train, y_train)
```

```
DummyClassifier(constant=None, random_state=42, strategy='uniform')
```

```
y_pred = random.predict(X_val)
mse["Random"] = mean_squared_error(y_val, y_pred)
```

```
for i in range(len(models)):
  print(models[i] + " vs Random")
  print(confidence_interval_between_models99(mse[models[i]], mse["Random"]))
```

```
Perceptron vs Random
[-0.0407  0.0882]
Logistic Regression vs Random
[-0.0307  0.0982]
Support Vector Machines vs Random
[0.1261 0.2514]
Decision Tree vs Random
[0.0625 0.19  ]
XG Boost vs Random
[0.1478 0.2722]
```

```
bestModels = [models[2], models[3], models[4]]
```

Conclusioni

I modelli basati su Perceptron e Logistic Regression non sono statisticamente migliori rispetto a quello Random. Tuttavia, i modelli che hanno mostrato score più alti già nelle precedenti analisi performano in modo significativamente migliore rispetto al modello casuale.

I modelli non lineari in particolare sono quelli che vanno meglio, questo a causa della scarsa correlazione delle feature. Inoltre, guardando altri notebook su Kaggle ho notato che tutti gli utenti che ottenevano score alti utilizzavano modelli non lineari.

I modelli migliori sono risultati essere:

```
pd.DataFrame([confidence_interval(accuracy(confusion_m[bestModels[i]]), len(X_val)
              index=bestModels,
              columns=["inf", "sup"])
```

	inf	sup
Support Vector Machines	0.6328	0.6981
Decision Tree	0.5694	0.6371
XG Boost	0.6545	0.7187