Lehrstuhlversuch im SS2020

Datenanalyse mit IceCube-Monte-Carlo-Simulationsdaten

Fabian Koch
fabian3.koch@tu-dortmund.de
Nils Breer
nils.breer@tu-dortmund.de
Nicole Schulte
nicole.schulte@tu-dortmund.de

Abgabe: xx.xx.2020

TU Dortmund – Fakultät Physik

Inhaltsverzeichnis

1 Theoretische Grundlagen

Kosmische Strahlung besteht hautpsächlich aus hochenergetischen Protonen, schweren Kernen, Myonen und Neutrinos. Die Komposition der geladenen kosmischen Strahlung hängt dabei vom Energiebereich ab. Es können dabei Energien bis zu 10^{20}eV erreicht werden. Die Energieverteilung folgt approximal einem Potenzgesetz der Form

$$\frac{\mathrm{d}\Phi}{\mathrm{d}E} = \Phi_0 E^{\gamma}$$

wobei γ der spektrale Index von etwa -2.7 für geladene Teilchen ist. Astrophysikalische Neutrinos stammen aus Quellen die auch Hadronen beschleunigen. Da Neutrinos einen sehr kleinen Wirkungsquerschnitt besitzen, durchdringen sie selbst dichte Staubwolken, welche für Photonen ein Hindernis sind. Außerdem werden Neutrinos nicht durch Magnetfelder abgelenkt und zeigen somit auf die Quelle und können so Informationen über das innere der Quelle liefern. Aufgrund von galaktischen Magnetfeldern ist es aber bislang nicht gelungen die Quellen der kosmischen Strahlung zu bestimmen. Wenn zur Beschleunigung eine Art Stoßbeschleunigung angenommen wird, also eine Art Fermibeschleunigung für Neutrinos, führt dies auf ein Potenzgesetz für den Neutrinofluss mit spektralen Index von $\gamma \approx -2$. Nun können Neutrinos und Myonen auch aus Wechselwirkungen in der Atmosphäre stammen, wo sie Zerfallsprodukte von Pionen und Kaonen sind. Da diese Mesonen eine vergleichsweise lange Lebensdauer haben verlieren sie vor ihrem Zerfall schon an Energie wodurch sich das Energiespektum einem Potenzgesetz proportional zu $E^{-3.7}$ gleicht. Die so entstandenen Neutrinos und Myonen nennt man konventionelle Neutrinos bzw. Myonen. Andererseits gibt es sogenannte prompte Neutrinos, welche entstehen wenn in hochenergetischen Wechselwirkungen kurzlebige schwere Hadronen erzeugt werden und ohne nennenswerten Energieverlust wieder zerfallen. Aus den (semi-)leptonischen Zerfällen stammen Neutrinos und Myonen welche das Energoespektrum der kosmischen Strahlung erben.

1.1 Messung von Neutrinos mit IceCube

Zur Messung wird die Analysetechnik "starting events" verwendet. Hierbei wird die äußersten Detektorabschnitte als Veto verwendet werden um atmosphärische Myonen zu verwerfen. Alle Neutrinoflavor werden hierbei berücksichtigt. Demnach sind die beiden größten Beiträge der Ereignisse aus dem neutralen Strom und der Elektron- bzw. Tauneutrinowechselwirkungen mittels des geladenen Stroms. Diese Ereignisse werden über Kaskaden im Detektor sichtbar und haben eine gute Energieauflösung, jedoch eine schlechte Winkelauflösung.
Myonen die durch den kompletten Detektor propagieren haben aufgrund der niedrigen Energiedeposition eine schlechtere Energieauflösung, ihre Spur kann jedoch gut rekonstruiert werden. Die Erde kann als Schild für atmosphärische Myonen verwendet werden, da diese durch sie zu größten Teilen absorbiert werden. Myonen die von unten eintreffen, müssen also aus Neutrinowechselwirkungen stammen. Auf den Zenitwinkel kann deswegen ein Schnitt gesetzt werden um Myonneutrinos und Myonen getrennt werden. Dadurch kann das Signal-Untergrund Verhältnis von $1 : 1 \cdot 10^6$ auf $1 : 1 \cdot 10^3$ verbessert werden.

2 Verwendete Methoden Maschinellen Lernens

2.1 Der kNN

2.2 Random Forrest

2.3 Jaccard Score

2.4 ROC-Kurven

3 Das IceCube-Experiment

Der IceCube Detektor dient der Detektion von hochenergetischen Neutrinos und Myonen und besteht aus drei Komponenten: Dem IceTop Detektor, dem In-Ice Detektor welcher die größte Komponente darstellt und dem DeepCore. Das Experiment befindet sich am geographischen Südpol und die Hauptdetektionsschicht ist zwischen 1450 m − 2450 m Tiefe in einer klaren Eisschicht. Mit Hilfe von Cherenkov Licht, welches mit 5160 DOMs[1] an 86 Strings detektiert werden kann, lassen sich hochenergetische geladene Teilchen detektieren. Ein schematischer Aufbau ist in Abbildung 1 gegeben.

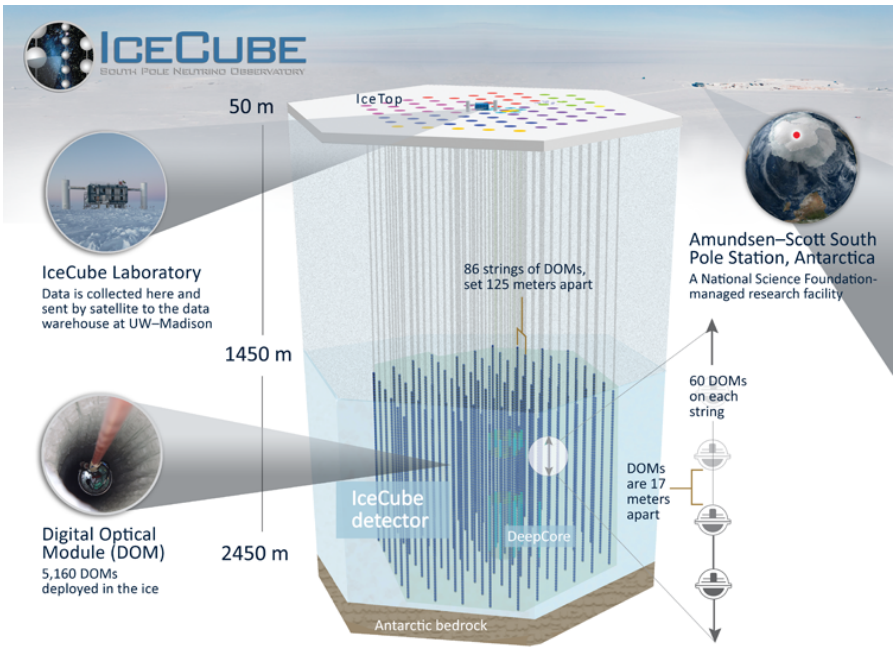


Abbildung 1: Schematischer Aufbau des IceCube Experiments [2].

Der DeepCore besteht aus sieben Kabeln, welche sich im Zentrum des In-Ice Detektors befinden. Die Energieschwelle im In-Ice Detektor beträgt 10 GeV. Sie ist demnach deutlich geringer als die Energieschwelle von 100 GeV des In-Ice Detektors. Das IceTop dient

[1] digital optical modules

als Luftschauer Experiment welches Cherenkov Licht in lichtdichten Eistanks detektiert. Außerdem kann es als Vetoregion für das In-Ice verwendet werden um gewisse Winkelbereiche auszuschließen.
Die Prozesse zur Detektion von Neutrinos geschehen über Sekundärteilchen aus den schwachen Wechselwirkungen mit den Kernen im Eis als geladener Strom

$$\nu_l(\bar{\nu}_l) + A \to l^{\mp} + X$$

oder neutraler Strom

$$\nu_l + A \to \nu_l + X\,.$$

Hierbei verursachen Elektronen eine sphärischen Schauer aufgrund des rapiden Energieverlustes. Myonen hingegen haben einen eher langsamen Energieverlust und können größere Distanzen überwinden und haben eine lange "Lichtspur" als Signatur. Tau-Leptonen haben eine ähnliche Signatur wie Elektronen aufgrund ihrer geringen Lebensdauer. Myonen erzeugen zu wenig Cherenkov Licht um detektiert zu werden aber sie generieren unter Anderem Photonen und e^+e^- Paare im Medium, welche selbst wieder schauern und weitere Elektron-Positron Paare erzeugen und von den PMT[2] detektiert werden, sofern die Geschwindigkeit der Sekundärteilchen schneller ist, als die Lichtgeschwindigkeit des Mediums.

4 Auswertung

Zu Beginn wurden aus den zur Vefügung gestellten Datensätzen alle Attribute die Monte Carlo Daten, Gewichte und Labels enthalten, entfernt. Außerdem wurden die Spalten entfernt, die ausschließlich den selben Wert, "Inf" oder "not a Number" enthalten. Abschließend wurde Signal und Hintergrund Samples auf Attribute überprüft, die nur in einem der beiden Samples auftreten und diese ebenfalls entfernt, sodass beide Samples die selben Attribute enthalten. Um später zwischen Hintergrund und Signal unterscheiden zu können, wurde an das Signal mit "0" gelabelt und der Hintergrund mit "1".
Im folgenden werden wir drei verschieden Klassifizierer auf eine Auswahl an Attributen testen. Es werden der `Naive-Bayes` Klassifizierer, der `RandomForestClassifier` und der `KNeighborsClassifier` mit Hilfe von `sklearn` verwendet. Vorab wurden mittels der `SelectKBest` Methode die 20 besten Attribute ermittelt. Die Güte der Attribute wurde mit dem `f_classif` ermittelt. Diese Attribute sind der beigefügten pdf des verwendeten jupyter Notebooks zu entnehmen. Aus diesen Attributen wurden Test- und Trainingsdatensätze extrahiert, mit welchen die obigen Klassifizierer nun getestet werden.
Zuerst wurde der `RandomForestClassifier` verwendet. Dieser basiert auf den binären Entscheidungsbäumen. Bei einem Entscheidungsbaum wird an jedem Knoten ein Schnitt in einer Variable durchgeführt und die daraus entstandenen Teilmengen werden in den beiden Ästen des Knotens wiederum durch Schnitte unterteilt, bis entweder eine bestimmte Tiefe des Baumes erreicht ist oder die Blätter nur Ereignisse einer Klasse enthalten. Um die Effekte des Übertrainierens zu minimieren, wird über ein Ensemble unterschiedlicher

[2]Photomultipliern

Entscheidungsbäume gemittelt. Die Bäume werden dabei jeweils auf unterschiedlichen Teilmengen des Trainingsadtensatzes trainiert. Die Attribute des Baumes an denen der beste Schnitt gesucht wird, werden zufällig ausgescucht. Dafür wurde ein Wald mit 100 Bäumen gewählt. Alle anderen Parameter wurde mit default initialisiert. Mit der Vorhersage des Klassifizierers wurden die Effizienz[3] und die Reinheit("precision") bestimmt. Außerdem wurde der Jaccard Score für unsere obige Attributsauswahl berechnet. Die Entsprechende ROC-Kurve ist Abbildung 2 zu entnehmen. Die Ergebnisse stehen in Tabelle 1.
Für den `KNeighborsClassifier` wurden 20 Nachbarn gewählt. Auch hier wird die Vorhersage und der Jaccard Score in Tabelle 1 dargestellt sowie die ROC-Kurve in Abbildung 2.
Zuletzt wurde der Naive-Bayes Klassifizierer, welcher dem Prinzip der bedingten Wahrscheinlichkeiten genügt, getestet. Die Ergebnisse befinden sich wieder in Tabelle 1 und der Abbildung 2.

Klassifizierer	Effizienz	Reinheit	Jaccard-Score
RandomForest	0.93425	0.92247	0.87762
KNeighborsClassifier	0.87913	0.87376	0.78463
Naive-Bayes	0.79763	0.75398	0.68410

Tabelle 1: Effizienz, Reinheit und Jaccard Score der drei verwendeten Klassifizierer.

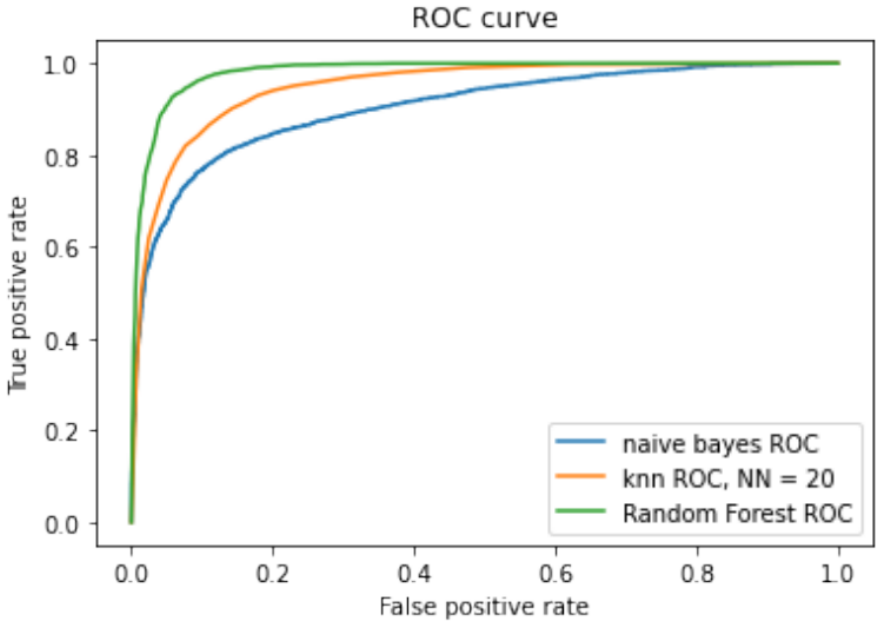


Abbildung 2: ROC-Kurven aller getesteten Klassifizierer.

[3]wir haben den "accuracy score" von sklearn verwendet da es keinen anderen gab

5 Diskussion

Im Allgemeinen ist festzuhalten, dass der Naive-Bayes Lerner die schlechteste Effizienz der drei Klassifizierer aufweist. Allerdings ist der Naive-Bayes Lerner relativ schnell und trifft gute Vorhersagen, wenn die Attribute unabhängig voneinander sind. Außerdem funktioniert er am besten mit kategorischen Attributen und weniger gut mit den hier verwendeten numerischen Eingaben. Außerdem wird der Naive-Bayes Lerner auch "schlechter Schätzer" genannt, weswegen der Output von Methoden wie `predict_proba` mit Bedacht verwendet werden sollten. Dies wird auch durch die ROC-Kurve deutlich. Dort verläuft die Kurve wesentlich flacher, die Fläche unterhalb der Kurve ist daher wesentlich geringer als bei den anderen beiden Klassifizierern. Zu beobachten ist ebenfalls der schlechte Jaccard-Score. Die Trennung zwischen Signal und Untergrund ist hier nicht gut erfolgt.
Der kNN-Klassifizierer, welcher auch als "Lazy lerner" bekannt ist gehört zu der Familie des überwachten maschinellen Lernens und wird oft als Maßstab für komplexere Lerner wie Support Vector Machines (SVM) verwendet. Der kNN ist relativ schnell für wenige Attribute doch leidet sehr unter dem Fluch der Dimensionalität. Bei 20 verwendeten Attributen und somit 20 nächsten Nachbarn, wird der Algorithmus sehr langsam und die Vorhersage wird mit zunehmender Attributszahl immer schlechter. Außerdem sollten die Attribute die gleiche Größenordnung besitzen da die Abstandsbestimmung mit der euklidischen Norm berechnet wurde. Um den Algorithmus noch effizienter zu machen, sollten hier die Daten vor dem Training normiert werden. Dennoch ist der kNN immer noch besser als der Naive-Bayes Lerner. Die Effizienz und die Reinheit sind bereits wesentlich besser als noch bei dem Naive-Bayes Lerner. Auch der Jaccard-Score zeigt bereits eine gute Trennung.
Der `RandomForestClassifier` ist auf dem genutzen Sample der beste Klassifizierer. Er kann die Entscheidungsbäume dekorrelieren um die auftretenden Korrelationen in den Attributen zu kontrollieren. Die Fehler bleiben vergleichsweise klein, da der Random Forest den Output jedes Baums verwendet und so Abweichungen minimiert. Hier könnte die Effizienz noch verbessert werden, indem eine andere Feature Selection verwendet wird, wie zum Beispiel eine Hauptkomponentenanalyse. Durch eine Variation der Anzahl an ausgewählten Attributen könnten eventuelle Overfitting Effekte gefunden werden. Dafür müssten die Effizienzen neuberechnet werden. Damit ließen sich eventuell Attribute finden, die die Effizienz verringern, obwohl sie zunächst von der Selektion als wichtig angesehen wurden. Allerdings sind die erreichte Effizienz und Reinheit bereits sehr gut. Die berechnete ROC- Kurve ähnelt dem eines perfekten Lerners schon sehr gut. Dieser würde eine senkrechte Linie nach oben und anschließend einen konstanten Wert von 1.0 bei der true positive rate aufweisen.

6 Anhang

Icecube

October 1, 2020

[1]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from skfeature.function.information_theoretical_based import MRMR

pd.set_option('display.max_columns', None)
```

[2]:
```
signal = pd.read_csv("signal.csv", sep = ";")
bkg = pd.read_csv("background.csv", sep = ";")
```

[3]:
```
signal = signal.drop(signal.filter(regex='MC').columns, axis=1)
signal = signal.drop(signal.filter(regex='Weight').columns, axis=1)
signal = signal.drop(signal.filter(regex='Corsika').columns, axis=1)
signal = signal.drop(signal.filter(regex='I3EventHeader').columns, axis=1)
signal = signal.drop(signal.filter(regex='end').columns, axis=1)
signal = signal.drop(signal.filter(regex='start').columns, axis=1)
signal = signal.drop(signal.filter(regex='time').columns, axis=1)
signal = signal.drop(signal.filter(regex='NewID').columns, axis=1)
signal = signal.drop('label', axis=1)
```

[4]:
```
bkg = bkg.drop(bkg.filter(regex='MC').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='Weight').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='Corsika').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='I3EventHeader').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='end').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='start').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='time').columns, axis=1)
bkg = bkg.drop(bkg.filter(regex='NewID').columns, axis=1)
bkg = bkg.drop('label', axis=1)
```

[5]:
```
signal.replace([np.inf, -np.inf], np.nan)
signal.dropna(axis = 'columns', inplace = True)
signal = signal.drop(signal.std()[(signal.std() == 0)].index, axis=1)
#signal.dropna(inplace = True)
```

```
bkg.replace([np.inf, -np.inf], np.nan)
bkg.dropna(axis = 'columns', inplace = True)
bkg = bkg.drop(bkg.std()[(bkg.std() == 0)].index, axis=1)
#bkg.dropna(inplace = True)
```

[6]:
```
bcol = bkg.columns
scol = signal.columns
```

[7]:
```
for att in scol:
    if att not in bcol:
        signal.drop(att, axis=1, inplace = True)

for att in bcol:
    if att not in scol:
        bkg.drop(att, axis=1, inplace = True)
```

[8]:
```
import scipy.io
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_validate
from sklearn import svm
from sklearn.metrics import jaccard_score
```

[9]:
```
sig_label = np.zeros(signal.shape[0])
bkg_label = np.ones(bkg.shape[0])
```

[10]:
```
combined_df = pd.concat([signal, bkg], ignore_index=True)
combined_label = np.append(sig_label, bkg_label)
```

[11]:
```
combined_df.insert(114, 'label', combined_label)
shuffled = combined_df.sample(frac = 1)
```

[12]:
```
y = shuffled['label']
X = shuffled.drop('label', axis=1)
```

[13]:
```
from sklearn import (
    ensemble, linear_model, neighbors, svm, tree, naive_bayes,
    gaussian_process, neural_network, dummy)
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.base import clone
from tqdm import tqdm
model = ensemble.RandomForestClassifier(n_estimators=100)
model.get_params()
from sklearn.model_selection import train_test_split
```

[14]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[15]:
```
from sklearn.feature_selection import SelectKBest, chi2, mutual_info_classif,␣
↪f_classif
# Anzahl der features die wir nehmen wollen
N_feat = 20
X_new = SelectKBest(score_func=f_classif, k=N_feat)
d_fit = X_new.fit(X_train, y_train)
# generiere scores die die güte des features angeben
scores = d_fit.scores_
# sortiere nach größe...
sorted_scores = sorted(scores, reverse=True)
args_max = np.argsort(scores)[::-1]
# print(args_max)
# lese die N_feat wichtigsten features aus und speichere sie weg
features = []
for i in range(N_feat):
    features.append(X.columns.tolist()[args_max[i]])
print(features)
# werfe aus den trainingsdaten und testdaten alle features bis auf die␣
↪wichtigsten raus
X_train = X_train.loc[:, features]
X_test = X_test.loc[:, features]
```

```
['LineFit_TTParams.lf_vel_z', 'HitStatisticsValues.z_travel',
'LineFit_TT.zenith', 'SplineMPEFitParams.rlogl', 'MuEXAngular4.zenith',
'SplineMPETruncatedEnergy_SPICEMie_AllDOMS_Muon.zenith',
'SplineMPEMuEXDifferential.zenith', 'SplineMPE.zenith',
'SplineMPETruncatedEnergy_SPICEMie_AllBINS_Muon.zenith',
'MPEFitHighNoise.zenith', 'MPEFitParaboloid.zenith',
'MPEFitParaboloidFitParams.zenith', 'SplineMPEDirectHitsA.n_dir_doms',
'SplineMPEDirectHitsA.n_dir_strings', 'SplineMPEDirectHitsC.dir_track_length',
'SplineMPETruncatedEnergy_SPICEMie_AllDOMS_MuEres.value',
'SplineMPEDirectHitsC.n_dir_doms', 'HitStatisticsValues.cog_z_sigma',
'HitStatisticsValues.z_sigma', 'NewAtt.DirectEllipse']
```

[16]:
```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import r2_score, roc_auc_score, roc_curve
# trainiere den lerner
model.fit(X_train, y_train)
# sage die labels vorher
y_pred = model.predict_proba(X_test)
y_pred = y_pred[:, 1]
fpr1, tpr1, thr1 = roc_curve(y_test, y_pred)
```

[17]:
```
#print(roc_auc_score(y_test, y_pred))
#print(r2_score(y_test, y_pred))
```

```
RFC_precision = precision_score(y_test, model.predict(X_test))
RFC_eff = accuracy_score(y_test, model.predict(X_test))
print('RFC accuracy score(sklearn) = ', RFC_eff)
print('RFC precision score(sklearn) = ', RFC_precision)
rfc_Jscore = jaccard_score(y_test, model.predict(X_test))
print('jaccard score, RFC: ', rfc_Jscore)
```

```
RFC accuracy score(sklearn) =  0.93425
RFC precision score(sklearn) =  0.9224749327463928
jaccard score, RFC:  0.8776174965100046
```

[18]:
```
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd
from scipy.spatial import distance
from sklearn.metrics.cluster import v_measure_score
from sklearn.metrics import accuracy_score

class KNN:
    def __init__(self, k):
        self.k = k


    def euc(self, a, b):
        return distance.euclidean(a, b)


    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train


    def predict(self, X):
        pred = []
        # row_count = 0
        for row in X: # iteriere durch jedes event
            # label = orte mit kleinstem abstand
            label = self.closest_points(row, self.X_train)
            # pred.append(prediction)
            N_sig = 0
            N_bg = 0
            for i in label:
                if self.y_train[i] == 0:
                    N_sig += 1
                else:
                    N_bg += 1
```

```
            if N_sig >= N_bg: # wenn label = signal ist dann 0 ( weil 0 =␣
↪signal heisst)
                pred.append(0)
            else:
                pred.append(1) # sonst 1 appenden da 1 = bg
            # if row_count % 10000 == 0:
                # print(row_count)
            # row_count += 1
        return pred


    def closest_points(self, row, X):
        index = [] # speichere hier die besten indizes
        distances = [] # alle distanzen
        distances = distance.cdist([row], X, 'euclidean')

        sort_dist = np.argsort(distances) # sortiere die distancen und gibt die␣
↪k kleinsten zurueck
        for j in range(self.k):
            index.append(sort_dist[0][j])

        return index
```

[19]:
```
nn = 20
knn = KNN(nn)
# daten muessen np arrays sein
X_trn = np.array(X_train)
y_trn = np.array(y_train)
X_t = np.array(X_test)
y_t = np.array(y_test)

knn_fit = knn.fit(X_trn, y_trn)
knn_pred = knn.predict(X_t)
```

[20]:
```
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd
from scipy.spatial import distance
from sklearn.metrics.cluster import v_measure_score
from sklearn.metrics import accuracy_score

knn_roc_score = roc_auc_score(y_test, knn_pred)
#print('roc_auc score (knn): ', knn_roc_score)

y_true = y_t.tolist()

tp = 0
```

```
fp = 0
fn = 0
tn = 0
for i in range(len(y_true)):
    if knn_pred[i] == y_true[i] and knn_pred[i] == 0:
        tp += 1
    if knn_pred[i] == 0 and y_true[i] == 1:
        fp += 1
    if knn_pred[i] == 1 and y_true[i] == 0:
        fn += 1
    if knn_pred[i] == y_true[i] and knn_pred[i] == 1:
        tn += 1

# print('tp: ', tp, 'fp: ', fp, 'fn: ', fn, 'tn:', tn)

Eff = tp / (tp + fn)
P = tp / (tp + fp)
S = tp / np.sqrt(tp + tn)
accuracy = (tp + tn) / (tp + fp + tn + fn)

print('for kNN:')
print('accuracy score(sklearn) = ', accuracy_score(knn_pred, y_true))
print('Eff: ', Eff)
print('Purity: ', P)
print('Signifikanz: ', S)
print('Accuracy: ', accuracy)
```

```
for kNN:
accuracy score(sklearn) =  0.879125
Eff:  0.8733515799950237
Purity:  0.8845766129032258
Signifikanz:  41.853984367007584
Accuracy:  0.879125
```

[21]:
```
# knn von sklearn:
from sklearn.neighbors import KNeighborsClassifier
knn_clf = KNeighborsClassifier(n_neighbors=nn)
knn_clf.fit(X_train, y_train)
PRED_knn = knn_clf.predict_proba(X_test)
PRED_knn = PRED_knn[:, 1]
fpr2, tpr2, thr2 = roc_curve(y_test, PRED_knn)
```

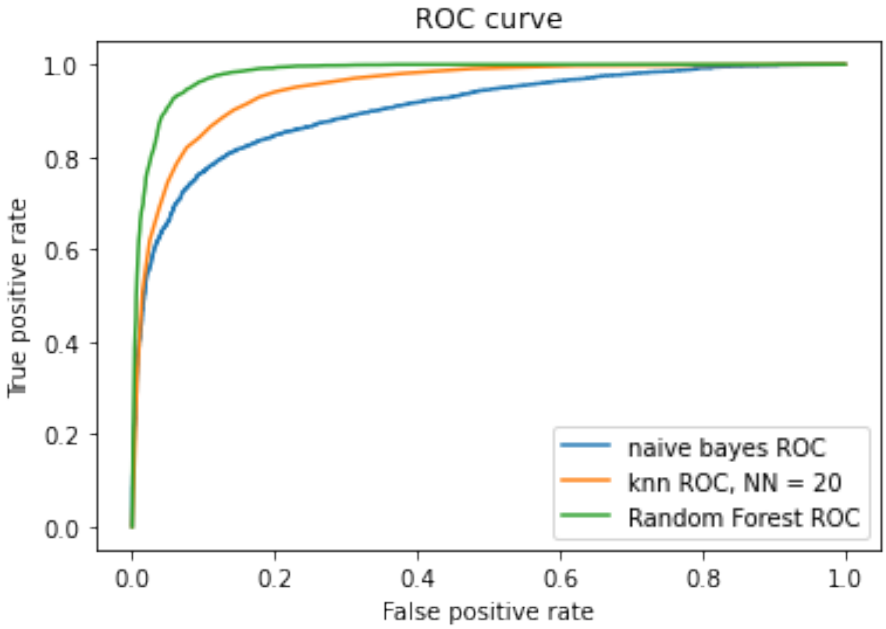
[22]:
```
knn_precision = precision_score(y_true, knn_clf.predict(X_test))
knn_eff = accuracy_score(y_true, knn_clf.predict(X_test))
print('KNN accuracy score(sklearn) = ', knn_eff)
print('KNN precision score(sklearn) = ', knn_precision)
knn_Jscore = jaccard_score(y_true, knn_clf.predict(X_test))
```

```
print('jaccard score, kNN: ', knn_Jscore)
```

```
KNN accuracy score(sklearn) =  0.879125
KNN precision score(sklearn) =  0.8737599206349206
jaccard score, kNN:  0.7846325167037862
```

[23]:
```
# naive bayes:
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
clf.fit(X_train, y_train)
NB_pred = clf.predict_proba(X_test)
NB_pred = NB_pred[:, 1]
fpr3, tpr3, thr3 = roc_curve(y_test, NB_pred)
```

[24]:
```
plt.figure(1)
plt.plot(fpr3, tpr3, label='naive bayes ROC')
plt.plot(fpr2, tpr2, label='knn ROC, NN = {}'.format(nn))
plt.plot(fpr1, tpr1, label='Random Forest ROC')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()
```



[25]:
```
NB_precision = precision_score(y_true, clf.predict(X_test))
NB_eff = accuracy_score(y_true, clf.predict(X_test))
print('NB accuracy score(sklearn) = ', NB_eff)
print('NB precision score(sklearn) = ', NB_precision)
NB_Jscore = jaccard_score(y_true, clf.predict(X_test))
print('jaccard score, NB: ', NB_Jscore)
```

```
NB accuracy score(sklearn) =  0.797625
NB precision score(sklearn) =  0.753978494623656
jaccard score, NB:  0.6840975609756098
```

Literatur

[1] IceCube Collaboration, *The IceCube Neutrino Observatory: instrumentation and online systems*, Journal of Instrumentation, JINST 12 P03012 (2017)

[2] IceCube Collaboration, Detektor, https://icecube.wisc.edu/science/icecube/detector