Contents

1 Executive Summary

In this report, I will follow the counterfactual explanations approach for explaining the behvaior of classification models. I will show how to use DiCE (Diverse Counterfactual Explanations) and ALIBI to provide “What if” cases by showing feature-perturbed versions of the same cases that would have had a different prediction.

2 Introduction

By identifying relevent features which may lead to stroke, medical professional can provide meedical advice to their patients. I plan to build a ANN and a Decision tree classifier in python to estimate the vulnerability of patients in having a stroke. In the last step of my analysis I will focus on ALIBI and DiCE to create the counterfactual explanations and explain the behvaior of classification models.

2.1 Note

This notebook should be run on "tnsf" enviornment with tensorflow and DiCE

A nice tuorial on this topic can also be founf here: https://www.youtube.com/watch?v=3bVGJZ5tHdg (https://www.youtube.com/watch?v=3bVGJZ5tHdg)

3 Loading the Dataset

In [1]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
import time
```

In [2]:
```
# Making a list of missing value types
missing_values = ["n/a", "na", "--"]
data1=pd.read_csv('Downloads/train_2v.csv', header=0, na_values = missing_values)
```

In [3]:
```
data1.head(5)
```

Out[3]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	30669	Male	3.0	0	0	No	children	Rural	95.12	18.0	NaN	0
1	30468	Male	58.0	1	0	Yes	Private	Urban	87.96	39.2	never smoked	0
2	16523	Female	8.0	0	0	No	Private	Urban	110.89	17.6	NaN	0
3	56543	Female	70.0	0	0	Yes	Private	Rural	69.04	35.9	formerly smoked	0
4	46136	Male	14.0	0	0	No	Never_worked	Rural	161.28	19.1	NaN	0

In [15]:
```
data1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 43400 entries, 0 to 43399
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   id                 43400 non-null  int64  
 1   gender             43400 non-null  object 
 2   age                43400 non-null  float64
 3   hypertension       43400 non-null  int64  
 4   heart_disease      43400 non-null  int64  
 5   ever_married       43400 non-null  object 
 6   work_type          43400 non-null  object 
 7   Residence_type     43400 non-null  object 
 8   avg_glucose_level  43400 non-null  float64
 9   bmi                41938 non-null  float64
 10  smoking_status     30108 non-null  object 
 11  stroke             43400 non-null  int64  
dtypes: float64(3), int64(4), object(5)
memory usage: 4.0+ MB
```

In [4]:
```
data1= data1.drop(columns=["id"], axis=1)
data1 = data1.dropna()
```

3.1 Data size and structure

In [157]:
```
data1.describe()
```

Out[157]:

	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke
count	29072.000000	29072.000000	29072.000000	29072.000000	29072.000000	29072.000000
mean	47.671746	0.111482	0.052146	106.403225	30.054166	0.018850
std	18.734490	0.314733	0.222326	45.268512	7.193908	0.135997
min	10.000000	0.000000	0.000000	55.010000	10.100000	0.000000
25%	32.000000	0.000000	0.000000	77.627500	25.000000	0.000000
50%	48.000000	0.000000	0.000000	92.130000	28.900000	0.000000
75%	62.000000	0.000000	0.000000	113.910000	33.900000	0.000000
max	82.000000	1.000000	1.000000	291.050000	92.000000	1.000000

In [11]:
```
data1.shape
```

Out[11]: (29072, 11)

3.2 Label Encoding

Contents

In [158]:
```
from sklearn.preprocessing import LabelEncoder
labels = data1.stroke
cat_columns_ix_ = {c: i for i, c in enumerate(data1.columns)
                                    if pd.api.types.is_object_dtype(data1[c])}
cat_columns  = [c for c in (data1.columns) if pd.api.types.is_object_dtype(data1[c])]
data_numeric = data1.copy() # categorical fields are label encoded but numerical
data_categorical = data1.copy() # categorical fields are label encoded, numeric but categorical type
data_ohe =  pd.get_dummies(data1,columns=cat_columns, sparse=False) # categorical fields are one hot encoded
for col in cat_columns:
    data_numeric[col] = (LabelEncoder().fit_transform(data_numeric[col]))
    data_categorical[col] = data_numeric[col].astype("category")
current_data = data_categorical
print("Available features: \n", list(data_categorical.columns))
print("Label Balance - [No Stroke, Stroke] : ", list(labels.value_counts()))
```

```
Available features:
 ['gender', 'age', 'hypertension', 'heart_disease', 'ever_married', 'work_type', 'Residence_type', 'avg_glucose_level', 'bm
i', 'smoking_status', 'stroke']
Label Balance - [No Stroke, Stroke] :  [28524, 548]
```

In [159]:
```
current_data.head(5)
```

Out[159]:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
1	1	58.0	1	0	1	2	1	87.96	39.2	1	0
3	0	70.0	0	0	1	2	0	69.04	35.9	0	0
6	0	52.0	0	0	1	2	1	77.59	17.7	0	0
7	0	75.0	0	1	1	3	0	243.53	27.0	1	0
8	0	32.0	0	0	1	2	0	77.67	32.3	2	0

4 Imbalanced Data

4.1 SMOTE

(Synthetic Minority Over-sampling Technique for Nominal and Continuous)

Unfortunately, a big drawback of most SMOTE and ADASYN algorithms is that they don't handle categorical features properly. Fortunately there is one variation of the SMOTE algorithm called 'SMOTE-NC' (Synthetic Minority Over-sampling Technique for Nominal and Continuous) that can deal with both categorical and continuous features.

In [160]:
```
from imblearn.over_sampling import SMOTENC
cat_cols =  ['gender','ever_married', 'work_type', 'Residence_type', 'hypertension', 'heart_disease', 'smoking_status']
labels= current_data['stroke']
features= current_data.loc[:, current_data.columns != 'stroke']
cat_cols_ind = []
for key in cat_cols:
    ind = features.columns.get_loc(key)
    cat_cols_ind.append(ind)
#Fit SMOTE-NC
smote_nc = SMOTENC(categorical_features=cat_cols_ind, random_state=42)
features_resampled, labels_resampled = smote_nc.fit_resample(features, labels)
labels_df=pd.DataFrame(data=labels_resampled)
```

In [161]:
```
X=features_resampled
Xcopy= X.copy()
for col in ['gender','ever_married', 'work_type', 'Residence_type', 'hypertension', 'heart_disease', 'smoking_status']:
    X[col] = X[col].astype('category')
y=labels_resampled
X["stroke"]=labels_df["stroke"]
```

5 Scaling

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
current_data[['age','stroke']] = scaler.fit_transform(current_data[['age','stroke']])
current_data[['bmi','stroke']]= scaler.fit_transform(current_data[['bmi','stroke']])
current_data[['avg_glucose_level','stroke']]= scaler.fit_transform(current_data[['avg_glucose_level','stroke']])
```

6 Feature Importance (Boruta)

6.1 Shadow Features

In Boruta, features do not compete among themselves. Instead, they compete with a randomized version of them. In practice, starting from X, another dataframe is created by randomly shuffling each feature. These permuted features are called shadow features. At this point, the shadow dataframe is attached to the original dataframe to obtain a new dataframe (X_boruta), which has twice the number of columns of X.

In [10]:
```
np.random.seed(101)
X_shadow = Xcopy.apply(np.random.permutation)
X_shadow.columns = ['shadow_' + feat for feat in Xcopy.columns]
### make X_boruta by appending X_shadow to X
X_boruta = pd.concat([Xcopy, X_shadow], axis = 1)
```

In [11]:
```
X_boruta.head(1)
```

Out[11]:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	shadow_gender
0	1	0.666667	1	0	1	2	1	0.139595	0.355311	1	0

6.2 Random Forest Classifier

Contents

Now, we take the importance of each original features and compare it with a threshold. This time, the threshold is defined as the highest feature importance recorded among the shadow features. When the importance of a feature is higher than this threshold, this is called a “hit”. The idea is that a feature is useful only if it’s capable of doing better than the best randomized feature.

In [12]:
```
from sklearn.ensemble import RandomForestClassifier
### fit a random forest (suggested max_depth between 3 and 7)
forest = RandomForestClassifier(n_estimators=50)
forest.fit(X_boruta, labels_df)
### store feature importances
feat_imp_X = forest.feature_importances_[:len(Xcopy.columns)]
feat_imp_shadow = forest.feature_importances_[len(Xcopy.columns):]
### compute hits
hits = feat_imp_X > feat_imp_shadow.max()
```

In [187]:
```
X_boruta.dtypes
```

Out[187]:
```
gender                    category
age                        float64
hypertension                 int64
heart_disease                int64
ever_married              category
work_type                 category
Residence_type            category
avg_glucose_level          float64
bmi                        float64
smoking_status            category
shadow_gender                int64
shadow_age                 float64
shadow_hypertension          int64
shadow_heart_disease         int64
shadow_ever_married          int64
shadow_work_type             int64
shadow_Residence_type        int64
shadow_avg_glucose_level   float64
shadow_bmi                 float64
shadow_smoking_status        int64
dtype: object
```

In [13]:
```
feat_imp_shadow
```

Out[13]:
```
array([0.00722357, 0.04520528, 0.00557117, 0.00436542, 0.00528522,
       0.01144058, 0.0074239 , 0.04786136, 0.04694423, 0.01147852])
```

In [14]:
```
feat_imp_shadow.max()
```

Out[14]:
```
0.047861360076629965
```

In [18]:
```
feat_imp_X
```

Out[18]:
```
array([0.01562295, 0.36834299, 0.02206534, 0.01612431, 0.04737771,
       0.0415712 , 0.02282208, 0.13625821, 0.10274236, 0.03427359])
```

In [15]:
```
hits
```

Out[15]:
```
array([False,  True, False, False, False, False, False,  True,  True,
       False])
```

The threshold is 0.047, thus 3 features made a hit, namely age, avg_glucose_level and bmi (respectively 0.36, 0.1 and 0.13). Apparently, we should drop the other features and get on with the seleted fearures (only for one run). What if instead it was just a lucky run for the selected variables?

6.3 Binomial Distribution

The maximum level of uncertainty about the feature is expressed by a probability of 50%, like tossing a coin. Since each independent experiment can give a binary outcome (hit or no hit), a series of n trials follows a binomial distribution.

In Boruta, there is not a hard threshold between a refusal and an acceptance area. Instead, there are 3 areas: an area of refusal (the red area): the features that end up here are considered as noise, so they are dropped; an area of irresolution (the blue area): Boruta is indecisive about the features that are in this area; an area of acceptance (the green area): the features that are here are considered as predictive, so they are kept.

6.4 Initialize Boruta

In [17]:
```
from boruta import BorutaPy
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(
   n_estimators=100
)
boruta = BorutaPy(
   estimator = forest,
   n_estimators = 'auto',
   max_iter = 100 # number of trials
)
### fit Boruta (it accepts np.array, not pd.DataFrame)
boruta.fit(np.array(Xcopy), np.array(labels_df))
### print results
green_area = Xcopy.columns[boruta.support_].to_list()
blue_area = Xcopy.columns[boruta.support_weak_].to_list()
print('features in the green area:', green_area)
print('features in the blue area:', blue_area)
```

```
features in the green area: ['age', 'ever_married', 'avg_glucose_level', 'bmi']
features in the blue area: []
```

The features stored in boruta.support_ are the ones that at some point ended up in the acceptance area, thus we should include them in your model. The features stored in boruta.support*weak* are the ones that Boruta didn’t manage to accept or refuse (blue area) and the choice is up to the data scientist: these features may be accepted or not depending on the use case.

In [193]:
```
Dataset= Xcopy[Xcopy.columns[boruta.support_]]
```

7 Counterfactual Explanations

7.1 Without Feature Selection

7.1.1 Using DiCE

In [18]:
```
import dice_ml
from dice_ml.utils import helpers
```

DiCE requires two inputs: a training dataset and a pre-trained ML model. It can also work without access to the full dataset.

7.1.1.1 Building the ML Model

In [23]:
```
from numpy.random import seed
seed(1)
import tensorflow as tf
import tensorflow.compat.v1 as tfc
#tf.random.set_seed(2)
from tensorflow import keras
# supress deprecation warnings from TF
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
sess = tfc.InteractiveSession()
```

In [194]:
```
dataset = X.copy()
d = dice_ml.Data(dataframe=dataset, continuous_features=["age", "avg_glucose_level", "bmi"], outcome_name='stroke')
train, test = d.split_data(d.normalize_data(d.one_hot_encoded_data))
X_train = train.loc[:, test.columns != 'stroke']
y_train = train.loc[:, test.columns == 'stroke']
X_test = test.loc[:, test.columns != 'stroke']
y_test = test.loc[:, test.columns == 'stroke']
```

In [218]:
```
ann_model = keras.Sequential()
ann_model.add(keras.layers.Dense(30, input_shape=(X_train.shape[1],), kernel_regularizer=keras.regularizers.l1(0.001), activ
ation=tfc.nn.relu))
ann_model.add(keras.layers.Dense(1, activation=tfc.nn.sigmoid))
ann_model.compile(loss='binary_crossentropy', optimizer=tfc.keras.optimizers.Adam(0.01), metrics=['accuracy'])
ann_model.fit(X_train, y_train, validation_split=0.20, epochs=100, verbose=0, class_weight={0:1,1:2})
```

Out[218]:
```
<tensorflow.python.keras.callbacks.History at 0x28552f8a608>
```

7.1.1.2 Loading the ML Model

In [219]:
```
backend = 'TF'+tfc.__version__[0] # TF1
m = dice_ml.Model(model=ann_model, backend=backend)
```

7.1.1.3 Initiate DiCE

In [220]:
```
exp = dice_ml.Dice(d, m)
```

7.1.1.4 query instance

in the form of a dictionary; keys: feature name, values: feature value

In [221]:
```
factual_sample = X_train.iloc[0, 0:].to_dict()
print("Counterfactual sample: {}".format(factual_sample))
```

```
Counterfactual sample: {'age': 0.640825092792511, 'avg_glucose_level': 0.8205905556678772, 'bmi': 0.3041047155857086, 'gende
r_0': 0.0, 'gender_1': 1.0, 'gender_2': 0.0, 'hypertension_0': 1.0, 'hypertension_1': 0.0, 'heart_disease_0': 1.0, 'heart_di
sease_1': 0.0, 'ever_married_0': 0.0, 'ever_married_1': 1.0, 'work_type_0': 0.0, 'work_type_1': 0.0, 'work_type_2': 1.0, 'wo
rk_type_3': 0.0, 'work_type_4': 0.0, 'Residence_type_0': 1.0, 'Residence_type_1': 0.0, 'smoking_status_0': 0.0, 'smoking_sta
tus_1': 0.0, 'smoking_status_2': 1.0}
```

7.1.1.5 Generate Counterfactuals

In [222]:
```
dice_exp = exp.generate_counterfactuals(factual_sample, total_CFs=4, desired_class="opposite")
```

```
Diverse Counterfactuals found! total time taken: 00 min 49 sec
```

7.1.1.6 Visualize the Resutls

In [223]:
```
dice_exp.visualize_as_dataframe()
```

```
Query instance (original outcome : 0)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	0	0.640825	0	0	0	0	0	0.820591	0.304105	0	0.000501

```
Diverse Counterfactual set (new outcome : 1)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	0	63.8	1	0	1	3	1	55.01	10.8	2	0.744
1	2	56.4	1	0	0	2	1	55.01	10.8	1	0.708
2	0	72.2	0	1	1	2	0	55.01	10.8	0	0.834
3	1	54.7	0	1	1	4	0	64.16	10.8	1	0.670

7.1.1.7 Highlight only the Changes

In [224]:
```
dice_exp.visualize_as_dataframe(show_only_changes=True)
```

```
Query instance (original outcome : 0)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	0	0.640825	0	0	0	0	0	0.820591	0.304105	0	0.000501

```
Diverse Counterfactual set (new outcome : 1)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	-	63.8	1	-	1	3	1	55.01	10.8	2	0.744
1	2	56.4	1	-	-	2	1	55.01	10.8	1	0.708
2	-	72.2	-	1	1	2	-	55.01	10.8	-	0.834
3	1	54.7	-	1	1	4	-	64.16	10.8	1	0.67

7.1.1.8 Customize Counterfactual Explanations

Selecting the features to vary & Trading off between proximity and diversity goals

Not all counterfactual explanations may be feasible for a user. In general, counterfactuals closer to an individual's profile will be more feasible. Diversity is also important to help an individual choose between multiple possible options. DiCE allows tunable parameters proximity_weight (default: 0.5) and diversity_weight (default: 1.0) to handle proximity and diversity respectively. We can increase the proximity weight and see how the counterfactuals change.

In [225]:
```
dice_exp = exp.generate_counterfactuals(factual_sample, total_CFs=4, desired_class="opposite", proximity_weight=1.5, diversi
ty_weight=1.0,
                                        features_to_vary=['gender','age','bmi','Residence_type'])
```

```
Only 0 (required 4) Diverse Counterfactuals found for the given configuation, perhaps try with different values of proximity
(or diversity) weights or learning rate... ; total time taken: 07 min 42 sec
```

Visualize the resutls

In [228]:
```
dice_exp.visualize_as_dataframe()
```

```
Query instance (original outcome : 0)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	0	0.640825	0	0	0	0	0	0.820591	0.304105	0	0.000501

```
Diverse Counterfactual set (new outcome : 1)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	0	61.2	0	0	0	0	1	55.01	10.8	0	0.025
1	2	70.4	0	0	0	0	0	55.01	10.8	0	0.027
2	1	45.9	0	0	0	0	1	55.01	10.8	0	0.008
3	2	37.1	0	0	0	0	1	55.01	16.7	0	0.004

Changing feature weights

feature_weight is a dictionary argument we can give for each numerical features to configure its difficulty to change the feature value for counterfactual explanations. By default, DiCE computes the inverse of MAD internally and divides the distance between continuous features by the MAD of the feature's values in the training set. A higher feature weight means that the feature is harder to change than others. For instance, one way is to use the mean absolute deviation from the median as a measure of relative difficulty of changing a continuous feature. Median Absolute Deviation (MAD) of a continuous feature conveys the variability of the feature, and is more robust than standard deviation as is less affected by outliers and non-normality. The inverse of MAD would then imply the ease of varying the feature and is hence used as feature weights in our optimization to reflect the difficulty of changing a continuous feature.

In [229]:
```
mads = d.get_mads(normalized=True)
# create feature weights
feature_weights = {}
for feature in mads:
    feature_weights[feature] = round(1/mads[feature], 2)
print(feature_weights)
```

```
{'age': 4.89, 'avg_glucose_level': 11.0, 'bmi': 21.53}
```

The above feature weights encode that changing age is approximately 5 times more difficult than changing categorical variables, and changing avg_glucose_level is approximately 11 times more difficult than changing age. Of course, this may sound odd, since a person cannot change their age. In this case, what it's reflecting is that there is a higher diversity in age values than avg_glucose_level values. Below we show how to over-ride these weights to assign custom user-defined weights.

By default, DiCE computes this internally and divides the distance between continuous features by the MAD of the feature's values in the training set.

Contents

In [230]:
```
# assigning equal weights
feature_weights = {'age': 1, 'avg_glucose_level': 1}
dice_exp = exp.generate_counterfactuals(query_instance, total_CFs=4, desired_class="opposite",
                                        feature_weights=feature_weights)
dice_exp.visualize_as_dataframe()
```

```
Diverse Counterfactuals found! total time taken: 00 min 32 sec
Query instance (original outcome : 1)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	1	58.0	1	0	1	2	1	87.959999	39.200001	1	0.776384

```
Diverse Counterfactual set (new outcome : 0)
```

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	1	10.0	1	1	1	3	1	232.76	33.8	1	0.000
1	0	10.0	1	0	1	2	1	87.85	10.8	1	0.001
2	1	62.7	1	0	1	0	1	55.01	39.2	2	0.262
3	1	14.2	1	0	0	2	0	87.91	61.2	1	0.000

7.1.2 Using Alibi

Alibi is a Python package designed to help explain the predictions of machine learning models and gauge the confidence of predictions. The focus of the library is to support the widest range of models using black-box methods where possible.

7.1.2.1 Importing Libraries

In [54]:
```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from alibi.explainers import AnchorTabular
```

7.1.2.2 Loading & Shuffling the Data

In [190]:
```
data1.head(1)
```

Out[190]:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
1	Male	58.0	1	0	Yes	Private	Urban	87.96	39.2	never smoked	0

In [195]:
```
Training_Data= dataset.loc[:, dataset.columns != 'stroke']
Training_Data_labels= dataset.loc[:, dataset.columns == 'stroke']
data_perm = np.random.permutation(np.c_[Training_Data, Training_Data_labels])
```

In [196]:
```
features= ['gender', 'age', 'hypertension', 'heart_disease', 'ever_married',
       'work_type', 'Residence_type', 'avg_glucose_level', 'bmi',
       'smoking_status']
catfeatures= ['gender','hypertension', 'heart_disease', 'ever_married', 'work_type', 'Residence_type', 'smoking_status']
numerical_features = ['age','avg_glucose_level', 'bmi']
for col in ['gender','ever_married', 'work_type', 'Residence_type', 'hypertension', 'heart_disease', 'smoking_status']:
    dataset[col] = dataset[col].astype('category')
categorical_features = [0, 2, 3, 4, 5, 6, 9]
```

In [197]:
```
data = data_perm[:,:-1]
labels = data_perm[:,-1]
data = data_perm.astype('int64')
labels =labels.astype('int64')
```

In [198]:
```
idx = 39000 # 70/30 split
X_train, Y_train = data[:idx,:-1], labels[:idx]
X_test, Y_test = data[idx+1:,:-1], labels[idx+1:]
```

In [200]:
```
Y_train
```

Out[200]:
```
array([0, 0, 0, ..., 0, 1, 0], dtype=int64)
```

7.1.2.3 One hot encoding of categorical features

In [396]:
```
categorical_transformer = Pipeline(steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

In [397]:
```
preprocessor = ColumnTransformer(transformers=[('cat', categorical_transformer, categorical_features)])
preprocessor.fit(X_train)
```

Out[397]:
```
ColumnTransformer(transformers=[('cat',
                                 Pipeline(steps=[('onehot',
                                                  OneHotEncoder(handle_unknown='ignore'))]),
                                 [0, 2, 3, 4, 5, 6, 9])])
```

7.1.2.4 Train Random Forest Model

In [398]:
```
np.random.seed(0)
clf = RandomForestClassifier(n_estimators=50)
clf.fit(preprocessor.transform(X_train), Y_train)
```

Out[398]:
```
RandomForestClassifier(n_estimators=50)
```

In [399]:
```
predict_fn = lambda x: clf.predict(preprocessor.transform(x))
print('Train accuracy: ', accuracy_score(Y_train, predict_fn(X_train)))
print('Test accuracy: ', accuracy_score(Y_test, predict_fn(X_test)))
```

```
Train accuracy:  0.7079487179487179
Test accuracy:  0.7049925195323322
```

7.1.2.5 Initialize and Fit Anchor Explainer for Tabular Data

In [401]:
```
explainer = AnchorTabular(predict_fn, features, seed=1)
```

Discretize the ordinal features into quartiles

In [402]:
```
explainer.fit(X_train, disc_perc=[25, 50, 75])
```

Out[402]:
```
AnchorTabular(meta={
    'name': 'AnchorTabular',
    'type': ['blackbox'],
    'explanations': ['local'],
    'params': {'seed': 1, 'disc_perc': [25, 50, 75]}
})
```

7.1.2.6 Getting an anchor

In [404]:
```
idx = 0
class_names = ['not stroke', 'stroke']
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])
```

```
Prediction:  stroke
```

In [405]:
```
explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

```
Anchor: work_type > 2.00 AND age > 60.00 AND 78.00 < avg_glucose_level <= 141.00 AND smoking_status <= 1.00 AND gender > 0.0
0 AND hypertension <= 0.00 AND ever_married <= 1.00
Precision: 0.96
Coverage: 0.24
```

when the anchor holds, the prediction should be the same as the prediction for this instance. We set the precision threshold to 0.95. This means that predictions on observations where the anchor holds will be the same as the prediction on the explained instance at least 95% of the time.

In [411]:
```
X_test[1000]
```

Out[411]:
```
array([ 0, 73,  0,  1,  1,  3,  0, 67, 35,  2], dtype=int64)
```

In [409]:
```
idx = 1000
class_names = ['not stroke', 'stroke']
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])

explanation = explainer.explain(X_test[idx], threshold=0.95)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

```
Prediction:  not stroke
Anchor: smoking_status > 1.00 AND avg_glucose_level <= 96.00 AND age <= 75.00 AND gender <= 0.00 AND hypertension <= 0.00 AN
D ever_married <= 1.00 AND Residence_type <= 1.00 AND bmi > 33.00 AND work_type > 2.00 AND heart_disease > 0.00
Precision: 0.99
Coverage: 0.60
```

Sometimes due to having an imbalnced dataset we may get the following results: "no anchor is found!"

This is due to the fact that during the sampling stage feature ranges corresponding to low-earners will be oversampled. It can also be fixed by producing balanced datasets to enable anchors to be found for either class.

7.2 With Feature Selection

7.2.1 Using DiCE

In [30]:
```
# we use the data after feature selection only
dataset = Dataset
dataset['stroke']= labels_df
```

In [32]:
```
d = dice_ml.Data(dataframe=dataset, continuous_features=["age", "avg_glucose_level", "bmi"], outcome_name='stroke')
train, test = d.split_data(d.normalize_data(d.one_hot_encoded_data))
X_train = train.loc[:, test.columns != 'stroke']
y_train = train.loc[:, test.columns == 'stroke']
X_test = test.loc[:, test.columns != 'stroke']
y_test = test.loc[:, test.columns == 'stroke']
```

In [33]:
```
ann_model = keras.Sequential()
ann_model.add(keras.layers.Dense(30, input_shape=(X_train.shape[1],), kernel_regularizer=keras.regularizers.l1(0.001), activ
ation=tfc.nn.relu))
ann_model.add(keras.layers.Dense(1, activation=tfc.nn.sigmoid))
ann_model.compile(loss='binary_crossentropy', optimizer=tfc.keras.optimizers.Adam(0.01), metrics=['accuracy'])
ann_model.fit(X_train, y_train, validation_split=0.20, epochs=100, verbose=0, class_weight={0:1,1:2})
```

Out[33]:
```
<tensorflow.python.keras.callbacks.History at 0x23080fc3a88>
```

In [34]:
```
backend = 'TF'+tfc.__version__[0] # TF1
m = dice_ml.Model(model=ann_model, backend=backend)
```

In [35]:
```
exp = dice_ml.Dice(d, m)
```

In [36]:
```
factual_sample = X_train.iloc[0, 0:].to_dict()
print("Counterfactual sample: {}".format(factual_sample))
```

```
Counterfactual sample: {'age': 0.7312705516815186, 'avg_glucose_level': 0.7388092875480652, 'bmi': 0.20786717534065247, 'eve
r_married_0': 0.0, 'ever_married_1': 1.0}
```

In [37]:
```
dice_exp = exp.generate_counterfactuals(factual_sample, total_CFs=4, desired_class="opposite")
```

```
Diverse Counterfactuals found! total time taken: 00 min 05 sec
```

In [38]:
```
dice_exp.visualize_as_dataframe()
```

```
Query instance (original outcome : 1)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.731271	0	0.738809	0.207867	0.822664

```
Diverse Counterfactual set (new outcome : 0)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.6220	0	0.5658	0.2088	0.214
1	0.2823	0	0.8080	0.2409	0.012
2	0.6612	0	0.7379	0.1202	0.258
3	0.3533	1	0.7379	0.1967	0.193

In [39]:
```
dice_exp.visualize_as_dataframe(show_only_changes=True)
```

```
Query instance (original outcome : 1)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.731271	0	0.738809	0.207867	0.822664

```
Diverse Counterfactual set (new outcome : 0)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.622	-	0.5658	0.2088	0.214
1	0.2823	-	0.808	0.2409	0.012
2	0.6612	-	0.7379	0.1202	0.258
3	0.3533	1	0.7379	0.1967	0.193

In [41]:
```
dice_exp = exp.generate_counterfactuals(factual_sample, total_CFs=4, desired_class="opposite", proximity_weight=1.5, diversity_weight=1.0,
                                        features_to_vary=['gender','age','bmi','ever_married'])
```

```
Diverse Counterfactuals found! total time taken: 00 min 26 sec
```

In [42]:
```
dice_exp.visualize_as_dataframe()
```

```
Query instance (original outcome : 1)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.731271	0	0.738809	0.207867	0.822664

```
Diverse Counterfactual set (new outcome : 0)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.5855	0	0.7388	0.1506	0.153
1	0.5029	0	0.7388	0.2435	0.092
2	0.6569	0	0.7388	0.1974	0.288
3	0.3931	1	0.7388	0.2081	0.266

In [44]:
```
mads = d.get_mads(normalized=True)
# create feature weights
feature_weights = {}
for feature in mads:
    feature_weights[feature] = round(1/mads[feature], 2)
print(feature_weights)
```

```
{'age': 4.85, 'avg_glucose_level': 11.26, 'bmi': 21.37}
```

In [46]:
```
feature_weights = {'age': 0.3, 'avg_glucose_level': 0.6 ,  'bmi': 1,  'ever_married': 0.1}
dice_exp = exp.generate_counterfactuals(factual_sample, total_CFs=4, desired_class="opposite",
                                        feature_weights=feature_weights)
dice_exp.visualize_as_dataframe()
```

```
Diverse Counterfactuals found! total time taken: 00 min 04 sec
Query instance (original outcome : 1)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.731271	0	0.738809	0.207867	0.822664

```
Diverse Counterfactual set (new outcome : 0)
```

	age	ever_married	avg_glucose_level	bmi	stroke
0	0.2197	1	0.0	0.0085	0.031
1	0.5148	0	0.0	1.0000	0.237
2	0.0000	0	1.0	0.0085	0.001
3	0.0000	1	1.0	1.0000	0.103

7.2.1.1 Intrepretation

The findings suggest: This person has less chance of having stroke if 1) the level of bmi goes down or the average glucose level goes down or he changes his marital status ((this is maybe unrealistic)) or he becomes younger (this is unrealistic).

7.2.2 Using Alibi

In [243]:
```
dataset=features_resampled.copy()
```

In [244]:
```
dataset = dataset.drop(columns= ['gender', 'hypertension', 'heart_disease',
       'work_type', 'Residence_type',
       'smoking_status'])
```

Contents

In [246]:
```
features= ['age', 'ever_married', 'avg_glucose_level', 'bmi']
catfeatures= ['ever_married']
numerical_features = ['age','avg_glucose_level', 'bmi']
for col in ['ever_married']:
    dataset[col] = (LabelEncoder().fit_transform(dataset[col]))
    dataset[col] = dataset[col].astype('category')
categorical_features = [1]
dataset[numerical_features]=dataset[numerical_features].astype('int64')
```

In [247]:
```
Training_Data= dataset.loc[:, dataset.columns != 'stroke']
Training_Data_labels= dataset.loc[:, dataset.columns == 'stroke']
data_perm = np.random.permutation(np.c_[Training_Data, Training_Data_labels])
```

In [248]:
```
data_perm
```

Out[248]:
```
array([[14, 0, 83, 20, 0],
       [53, 1, 86, 23, 1],
       [40, 1, 80, 26, 0],
       ...,
       [81, 1, 110, 21, 1],
       [66, 1, 80, 29, 1],
       [41, 1, 216, 36, 0]], dtype=object)
```

In [249]:
```
data = data_perm[:,:-1]
labels = data_perm[:,-1]
data = data_perm.astype('int64')
labels =labels.astype('int64')
```

In [250]:
```
idx = 39000 # 70/30 split
X_train, Y_train = data[:idx,:-1], labels[:idx]
X_test, Y_test = data[idx+1:,:-1], labels[idx+1:]
```

In [251]:
```
Y_test
```

Out[251]:
```
array([1, 1, 0, ..., 1, 1, 0], dtype=int64)
```

In [252]:
```
categorical_transformer = Pipeline(steps=[('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

In [253]:
```
preprocessor = ColumnTransformer(transformers=[('cat', categorical_transformer, categorical_features)])
preprocessor.fit(X_train)
```

Out[253]:
```
ColumnTransformer(transformers=[('cat',
                                 Pipeline(steps=[('onehot',
                                                  OneHotEncoder(handle_unknown='ignore'))]),
                                 [1])])
```

In [254]:
```
np.random.seed(0)
clf = RandomForestClassifier(n_estimators=50)
clf.fit(preprocessor.transform(X_train), Y_train)
#clf.fit( X_train,Y_train)
```

Out[254]:
```
RandomForestClassifier(n_estimators=50)
```

In [221]:
```
predict_fn = lambda x: clf.predict(preprocessor.transform(x))
print('Train accuracy: ', accuracy_score(Y_train, predict_fn(X_train)))
print('Test accuracy: ', accuracy_score(Y_test, predict_fn(X_test)))
```

```
Train accuracy:  0.6265641025641026
Test accuracy:  0.6205463511941043
```

In [255]:
```
explainer = AnchorTabular(predict_fn, features, seed=1)
```

Discretize the ordinal features into quartiles

In [259]:
```
explainer.fit(X_train, disc_perc=[25, 50, 75])
```

Out[259]:
```
AnchorTabular(meta={
    'name': 'AnchorTabular',
    'type': ['blackbox'],
    'explanations': ['local'],
    'params': {'seed': 1, 'disc_perc': (25, 50, 75)}
})
```

In [260]:
```
idx = 100
class_names = ['not stroke', 'stroke']
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])
```

```
Prediction:  stroke
```

In [261]:
```
explanation = explainer.explain(X_test[idx], threshold=0.4)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

```
Anchor:
Precision: 0.78
Coverage: 1.00
```

When the anchor holds, the prediction should be the same as the prediction for this instance. We set the precision threshold to 0.7. This means that predictions on observations where the anchor holds will be the same as the prediction on the explained instance at least 70% of the time.

In [242]:
```
X_test[100]
```

Out[242]:
```
array([ 80,   1, 107,  21], dtype=int64)
```

In [241]:
```
idx = 100
class_names = ['not stroke', 'stroke']
print('Prediction: ', class_names[explainer.predictor(X_test[idx].reshape(1, -1))[0]])

explanation = explainer.explain(X_test[idx], threshold=0.3)
print('Anchor: %s' % (' AND '.join(explanation.anchor)))
print('Precision: %.2f' % explanation.precision)
print('Coverage: %.2f' % explanation.coverage)
```

```
Prediction:  stroke
Anchor:
Precision: 0.85
Coverage: 1.00
```

7.2.2.1 Intrepretation

It is possible for the anchor algorithms to return an empty anchor. The interpretation here is that any feature/word/superpixel could act as an anchor as the sampling procedure couldn't produce examples of a different class, therefore there is no particularly important subset of features to produce the same prediction. We should document this in detail as well as decide if returning the "empty anchor" makes the most sense in such cases.