Tutorial

June 12, 2019

1 Introduction to the class Nescience

In this tutorial we are going to see how to use the class "Nescience", in order to compute the nescience of a model and a dataset, and to compute the individual terms that compose the nescience, that is, miscoding, inaccuracy and surfeit.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pylab import rcParams
```

[2]:
```
rcParams['figure.figsize'] = 20, 10
```

For the examples we are going to use the breast cancer dataset. Mind that in the current version the Nescience class can be applied only in case of classification problems with continuous features.

[3]:
```
from sklearn.datasets import load_breast_cancer
```

We will apply the nescience class to decision tree classifier models and multilayer perceptron models.

[4]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.neural_network import MLPClassifier
```

Finally we import the Nescience class.

[5]:
```
from Nescience.Nescience import *
```

1.1 Miscoding

Target Conditional Complexity The target conditional complexity measures the effort required to encode the target variable assuming the knowledge of a feature. The higher this value, the better, since that means the feature contains more information about the target.

We will use a synthetic dataset to show how this method works. We will randomly generate four clouds of points classified according to ten features among which only four are relevant.

[6]:
```
from sklearn.datasets.samples_generator import make_classification
```

[7]:
```
X, y = make_classification(n_samples=1000, n_features=20, n_informative=4,␣
↪n_redundant=0, n_repeated=0, n_classes=4, n_clusters_per_class=1,␣
↪weights=None, flip_y=0, class_sep=1.0, hypercube=True, shift=0.0, scale=1.0,␣
↪shuffle=True, random_state=1)
```

We have to initialize the class Nescience with the dataset we are going to use.

[8]:
```
nescience = Nescience(X, y, verbose=True)
```

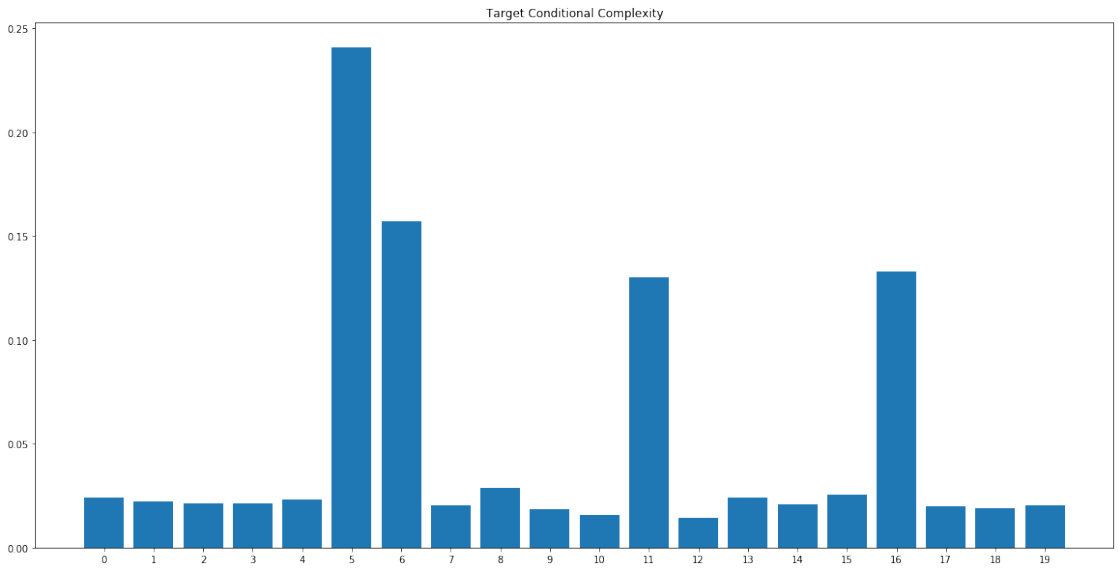
```
Computing optimal codes ... done!
Computing miscoding ... done!
```

Let's get the target conditional complexity with respect to all the features.

[9]:
```
tcc = nescience.targetconditionalcomplexity()
tcc
```

[9]:
```
array([0.02390238, 0.02232701, 0.02129829, 0.02132389, 0.02317516,
       0.24086956, 0.15724648, 0.02033484, 0.02859811, 0.01850099,
       0.01562849, 0.13001501, 0.01448534, 0.02431025, 0.02073056,
       0.02530215, 0.13286182, 0.01971741, 0.01882615, 0.02054609])
```

[10]:
```
plt.bar(x=np.arange(0, 20), height=tcc, tick_label=np.arange(0, 20))
plt.title("Target Conditional Complexity")
plt.show()
```



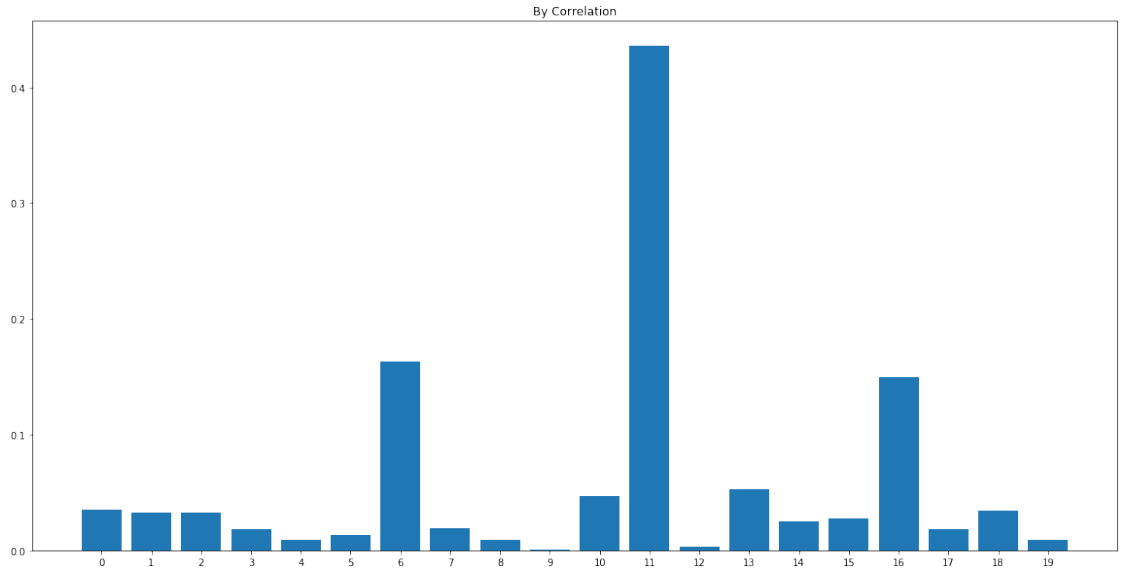
There are clearly four features that have some "predictive" power over the target variable. A value close to 1 means more predictive power.

Let's compare with a classical correlation schema.

[11]:
```
df = pd.DataFrame(X)
df['y'] = y
corr = df.corr()
```

[12]:
```
plt.bar(x=np.arange(0, 20), height=abs(corr['y'][:-1].values), tick_label=np.
 ↪arange(0, 20))
plt.title("By Correlation")
```

[12]:
```
Text(0.5,1,'By Correlation')
```



In this case, there are three clear features correlated with the target variable, however, it is not clear if there is a fourth one (and it should).

Model Miscoding Now, let's compute the miscoding of a trained model. The miscoding of a model measures the "relevance" of the collection of features used in the model to predict the target variable.

[13]:
```
data = load_breast_cancer()
X = data.data
y = data.target
```

[14]:
```
tree = DecisionTreeClassifier(min_samples_leaf=5)
tree.fit(X, y)
```

[14]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

In order to do that we need the list of features in use.

[15]:
```
attr_in_use = np.zeros(X.shape[1], dtype=int)
features = set(tree.tree_.feature[tree.tree_.feature >= 0])
for i in features:
    attr_in_use[i] = 1
```

Let's intitallize the Nescience class with this new dataset.

[16]:
```
nescience = Nescience(X, y, verbose=True)
```

```
Computing optimal codes ... done!
Computing miscoding ... done!
```

[17]:
```
nescience.miscoding(attr_in_use)
```

[17]:
```
0.5920245086169342
```

It seeems that the model is not using all the relevant attributes. Let's see which are the attributes in use.

[18]:
```
print(np.unique(data.feature_names[attr_in_use]))
```

```
['mean radius' 'mean texture']
```

[19]:
```
print(np.unique(data.feature_names))
```

```
['area error' 'compactness error' 'concave points error' 'concavity error'
 'fractal dimension error' 'mean area' 'mean compactness'
 'mean concave points' 'mean concavity' 'mean fractal dimension'
 'mean perimeter' 'mean radius' 'mean smoothness' 'mean symmetry'
 'mean texture' 'perimeter error' 'radius error' 'smoothness error'
 'symmetry error' 'texture error' 'worst area' 'worst compactness'
 'worst concave points' 'worst concavity' 'worst fractal dimension'
 'worst perimeter' 'worst radius' 'worst smoothness' 'worst symmetry'
 'worst texture']
```

1.2 Inaccuracy

The inaccuracy of a model, according to the theory of nescience, is the effort, measured as the length of a computer program, to fix the errors of the model. In practice, inaccuracy is approximated as the compressed version of the errors.

[20]:
```
data = load_breast_cancer()
X = data.data
y = data.target
```

[21]:
```
tree = DecisionTreeClassifier(min_samples_leaf=5)
tree.fit(X, y)
```

[21]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

[22]:
```
attr_in_use = np.zeros(X.shape[1], dtype=int)
features = set(tree.tree_.feature[tree.tree_.feature >= 0])
for i in features:
    attr_in_use[i] = 1
```

[23]:
```
errors = list()
pred = tree.predict(X)
for i in np.arange(X.shape[0]):
    if pred[i] != y[i]:
```

```
        new_error = list(X[i][np.where(attr_in_use)])
        new_error.append(y[i])
        errors.append(new_error)
```

[24]:
```
nescience = Nescience(X, y, verbose=True)
```

```
Computing optimal codes ... done!
Computing miscoding ... done!
```

[25]:
```
nescience.inaccuracy(np.array(errors), attr_in_use)
```

[25]: 0.008371934667309589

Let's Compare with the score of the model:

[26]:
```
1 - tree.score(X, y)
```

[26]: 0.02284710017574687

Let's see what happens if we make one hundred times the same error (if we do that in case of sklearn, the error will increase by 0.18).

[27]:
```
errors_2 = errors.copy()
for i in np.arange(100):
    errors_2.append(errors[0])
```

[28]:
```
nescience.inaccuracy(np.array(errors_2), attr_in_use)
```

[28]: 0.019566421799461907

The theory of nescience states that making one hundred times the same error is not that bad. Let's see what happens if we make one hundred different errors.

[29]:
```
errors_3 = errors.copy()
for i in np.arange(100):
    rnd = np.random.randint(2)
    errors_3.append(errors[rnd])
```

[30]:
```
nescience.inaccuracy(np.array(errors_3), attr_in_use)
```

[30]: 0.05006121290727005

Making one hundred different errors is worse than making one hundred times the same error.

1.3 Surfeit

Surfeit tell us how far we are from having the shortest possible model for this dataset. Surfeit, being a non-computable quantity, has to be approximated in practice. The approximation will be based on the redudancy of the model.

[31]:
```
data = load_breast_cancer()
X = data.data
y = data.target
```

[32]:
```
tree = DecisionTreeClassifier(min_samples_leaf=5)
tree.fit(X, y)
```

[32]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

First, we need a function that encode our model as a string.

[33]:
```
def tree2string(estimator):

    n_nodes        = estimator.tree_.node_count
    children_left  = estimator.tree_.children_left
    children_right = estimator.tree_.children_right
    feature        = estimator.tree_.feature
    threshold      = estimator.tree_.threshold

    node_depth = np.zeros(shape=n_nodes, dtype=np.int64)
    is_leaves  = np.zeros(shape=n_nodes, dtype=bool)

    stack = [(0, -1)]  # seed is the root node id and its parent depth

    while len(stack) > 0:

        node_id, parent_depth = stack.pop()
        node_depth[node_id] = parent_depth + 1

        # If we have a test node
        if (children_left[node_id] != children_right[node_id]):
            stack.append((children_left[node_id], parent_depth + 1))
            stack.append((children_right[node_id], parent_depth + 1))
        else:
            is_leaves[node_id] = True

    string = ""

    for i in range(n_nodes):

        if is_leaves[i]:
            string = string + str("%snode=%s leaf node." % (node_depth[i] *␣
↪"\t", i))
        else:
            string = string + ("%snode=%s test node: go to node %s if X[:, %s]␣
↪<= %s else to "
                               "node %s."
                               % (node_depth[i] * "\t",
                               i,
                               children_left[i],
```

```
                                feature[i],
                                threshold[i],
                                children_right[i],
                                ))

            string = string + "\n"

        return string
```

[34]:
```
print(tree2string(tree))
```

```
node=0 test node: go to node 1 if X[:, 20] <= 16.795000076293945 else to node
22.
        node=1 test node: go to node 2 if X[:, 27] <= 0.13580000400543213 else
to node 13.
                node=2 test node: go to node 3 if X[:, 10] <= 0.6430999934673309
else to node 12.
                        node=3 test node: go to node 4 if X[:, 23] <= 785.75
else to node 9.
                                node=4 test node: go to node 5 if X[:, 21] <=
33.35000038146973 else to node 6.
                                        node=5 leaf node.
                                        node=6 test node: go to node 7 if X[:,
21] <= 35.04999923706055 else to node 8.
                                                node=7 leaf node.
                                                node=8 leaf node.
                                node=9 test node: go to node 10 if X[:, 23] <=
809.0500183105469 else to node 11.
                                        node=10 leaf node.
                                        node=11 leaf node.
                        node=12 leaf node.
                node=13 test node: go to node 14 if X[:, 21] <=
25.670000076293945 else to node 19.
                        node=14 test node: go to node 15 if X[:, 23] <=
805.8999938964844 else to node 18.
                                node=15 test node: go to node 16 if X[:, 5] <=
0.156700000166893 else to node 17.
                                        node=16 leaf node.
                                        node=17 leaf node.
                                node=18 leaf node.
                        node=19 test node: go to node 20 if X[:, 7] <=
0.054099999368190765 else to node 21.
                                node=20 leaf node.
                                node=21 leaf node.
        node=22 test node: go to node 23 if X[:, 21] <= 19.90999984741211 else
to node 26.
                node=23 test node: go to node 24 if X[:, 27] <=
0.1454000025987625 else to node 25.
```

```
                        node=24 leaf node.
                        node=25 leaf node.
                    node=26 test node: go to node 27 if X[:, 26] <=
0.190700002014637 else to node 28.
                        node=27 leaf node.
                        node=28 leaf node.
```

Ideally, our encoding should provide a valid python function. For the sake of demonstration purposes, we will use the above encoding.

[35]:
```
nescience = Nescience(X, y, verbose=True)
```

```
Computing optimal codes ... done!
Computing miscoding ... done!
```

[36]:
```
nescience.redundancy(tree2string(tree))
```

[36]:
```
0.7303149606299213
```

Next, do the same for the case of a multi-layer perceptron neural network

[37]:
```
def nn2string(nn):

    # Header
    string = "def NN(X):\n"

    # Parameters
    for i in np.arange(len(nn.coefs_)):
        string = string + "    W" + str(i) + " = " + str(nn.coefs_[i]) + "\n"
        string = string + "    b" + str(i) + " = " + str(nn.intercepts_[i]) +␣
↪"\n"

    # Computation
    for i in np.arange(len(nn.coefs_) - 1):
        string = string + "    Z" + str(i) + " = np.matmul(W" + str(i) + ", X) +␣
↪b" + str(i) + "\n"
        string = string + "    A" + str(i) + " = np.tanh(Z" + str(i) + ")\n"

    i = len(nn.coefs_)
    string = string + "    Z" + str(i) + " = np.matmul(W" + str(i) + ", A" +␣
↪str(i) + ") + b2\n"
    string = string + "    A" + str(i) + " = self._sigmoid(Z" + str(i) + ")\n"

    # Predictions
    string = string + "    predictions = A" + str(i) + " > 0.5\n\n"

    string = string + "    return predictions\n"

    return string
```

[38]:
```
nn = MLPClassifier(hidden_layer_sizes = [5, 5],
                   activation         = "relu",
                   learning_rate      = "constant",
                   learning_rate_init = 0.01,
                   solver             = "sgd",
                   alpha              = 0,
                   max_iter           = 100,
                   tol                = 0)
```

[39]:
```
nn.fit(X, y)
```

[39]:
```
MLPClassifier(activation='relu', alpha=0, batch_size='auto', beta_1=0.9,
              beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes=[5, 5], learning_rate='constant',
              learning_rate_init=0.01, max_iter=100, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
              random_state=None, shuffle=True, solver='sgd', tol=0,
              validation_fraction=0.1, verbose=False, warm_start=False)
```

[40]:
```
print(nn2string(nn))
```

```
def NN(X):
    W0 = [[-2.81920432e-01  2.19592504e-01  1.75154976e-01  4.42013585e-02
   1.42498446e-01]
 [-6.26524258e-01 -2.03245572e-01  2.05092369e-01 -3.89630722e-01
  -2.01655469e-01]
 [-2.68940243e+00 -3.37916209e-01 -2.22730850e-01 -8.52894955e-01
  -4.27579057e-01]
 [-1.28653313e+01 -1.05849318e-01 -4.09080822e-01 -5.54218365e+00
  -8.49206583e-01]
 [ 5.13903842e-02 -3.50802314e-01 -3.95794652e-01  2.16950763e-01
   1.12009543e-02]
 [ 1.94345211e-01 -2.44673103e-01  1.63784172e-01  2.50287338e-01
  -2.57373601e-01]
 [ 3.89822971e-01 -1.59138757e-01  2.04751511e-01 -3.80761301e-01
  -2.53544860e-01]
 [-1.53405874e-01 -3.40619759e-01  3.24229917e-01  1.49065628e-01
  -2.51790981e-01]
 [-1.16787968e-01 -6.94427118e-02  5.89907294e-03 -1.43841488e-01
  -2.10121029e-01]
 [-3.26809921e-01  4.09035431e-01 -2.00619301e-01 -3.11008865e-01
  -3.42594250e-01]
 [ 2.66015109e-01 -3.06932935e-01  1.90729030e-01  2.69380586e-01
   2.09064386e-01]
 [-1.46815584e-01  3.91659560e-01  8.82679843e-02  1.18322232e-01
   3.11197947e-02]
 [ 1.02319636e-01 -1.83787200e-01 -2.70697040e-01  2.41337778e-01
  -2.44264533e-01]
 [-2.73511747e-01 -1.91653277e-01 -7.32037565e-03 -1.73541380e-01
```

```
 -2.60346801e-01]
[ 2.09759007e-01  3.45625367e-01 -2.67431951e-02 -1.55444895e-02
 -2.67252878e-01]
[-3.51533269e-01  2.53200140e-01  3.01767076e-01  3.25806522e-01
 -2.43997490e-01]
[ 1.35972214e-01 -3.41531703e-01 -8.96890037e-02 -4.11577266e-01
  3.63239913e-01]
[-1.87227515e-01 -1.63851583e-01 -7.80798793e-02 -3.57639024e-01
  3.78752297e-01]
[-2.54277019e-01  3.45595597e-02  3.64067432e-01 -2.83416793e-01
 -3.84976210e-02]
[-2.25923625e-01  1.83456039e-01  3.79331262e-01  3.85213987e-01
  3.64495012e-01]
[-1.56839413e-01 -3.64271197e-01  2.54256609e-01 -4.09580684e-01
 -9.87473549e-02]
[-4.45831730e-01  3.92103014e-01 -1.31868753e-01 -4.22837295e-01
  3.27978404e-01]
[-2.21032700e+00  3.62593439e-02 -5.70546031e-02 -8.03758974e-01
  2.00947545e-01]
[-1.63313955e+01 -2.45036840e-01 -3.07609566e-02 -7.90682756e+00
 -1.33301037e+00]
[-2.07505415e-01  3.66273012e-01  3.59403402e-01 -2.65207702e-03
  1.78303223e-01]
[ 3.02252523e-01  1.76358606e-01 -2.06966260e-01  1.61043402e-01
 -8.44452642e-02]
[ 3.24358545e-01 -3.25422599e-01 -1.12402901e-01 -3.13842718e-01
  1.95667287e-01]
[-3.16950022e-01  1.16473150e-01 -1.57253067e-01  1.40373674e-01
 -1.47435832e-01]
[-4.82426008e-02  1.37259391e-01  7.96204518e-02  2.08117032e-01
 -8.22580585e-02]
[-3.29958061e-01 -3.14792650e-01 -3.76061013e-01 -1.10748134e-01
  2.63409353e-01]]
   b0 = [-0.17112344 -0.04283281 -0.22563885  0.27677194 -0.40831204]
   W1 = [[  7.17108472  -9.11949849  -0.59621921  -0.19902993 -18.56919095]
[  0.62361371   0.71326158  -0.57962901  -0.44719908   0.68716421]
[ -0.73104992  -0.28711344  -0.19421397  -0.42897953   0.56773257]
[ -0.49685323  -0.52141855 -15.37719975  -0.76795533  -0.56421118]
[ -0.1722854    0.51548676  -0.34968296  -0.33441341  -0.22013223]]
   b1 = [-0.05047012 -0.95579354 -0.05155672 -0.65892115 -0.65368957]
   W2 = [[ 7.96526423]
[17.21865971]
[-3.25346716]
[-0.71916148]
[13.19158482]]
   b2 = [0.61266771]
   Z0 = np.matmul(W0, X) + b0
   A0 = np.tanh(Z0)
```

```
    Z1 = np.matmul(W1, X) + b1
    A1 = np.tanh(Z1)
    Z3 = np.matmul(W3, A3) + b2
    A3 = self._sigmoid(Z3)
    predictions = A3 > 0.5

    return predictions
```

Ideally we should not relay in the numpy library, since what we are computing is the conditional surfeit of a model given the numpy library. But let's do that for the sake of demo.

[41]:
```
nescience.redundancy(nn2string(nn))
```

[41]:
```
0.5608747044917257
```

1.4 Nescience

Nescience is a measure of how much we do not know about the problem at hand given a dataset and a model. It is a function of miscoding, inaccuracy and surfeit.

[42]:
```
data = load_breast_cancer()
X = data.data
y = data.target
```

[43]:
```
tree = DecisionTreeClassifier(min_samples_leaf=5)
tree.fit(X, y)
```

[43]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

Features in use.

[44]:
```
attr_in_use = np.zeros(X.shape[1], dtype=int)
features = set(tree.tree_.feature[tree.tree_.feature >= 0])
for i in features:
    attr_in_use[i] = 1
```

List of errors.

[45]:
```
errors = list()
pred = tree.predict(X)
for i in np.arange(X.shape[0]):
    if pred[i] != y[i]:
        new_error = list(X[i][np.where(attr_in_use)])
        new_error.append(y[i])
        errors.append(new_error)
errors = np.array(errors)
```

Model encoded as string.

[46]:
```
model_str = tree2string(tree)
```

Compute the nescience of the problem given a dataset and a model.

[47]:
```
nescience = Nescience(X, y, verbose=True)
```

```
Computing optimal codes ... done!
Computing miscoding ... done!
```

[48]:
```
nescience.nescience(attr_in_use, errors, model_str)
```

[48]:
```
0.024262130524488314
```

1.5 Optimal Tree

Lets see how we can apply the concept of nescience to find an optimal value for one of the hyper-parameters of decision trees. Please mind that this approach is different of the approach used in the NescienceDecisionTreeClassifer algorithm of the Nescience package.

[49]:
```
data = load_breast_cancer()
X = data.data
y = data.target
```

[50]:
```
nescience = Nescience(X, y, verbose=True)
```

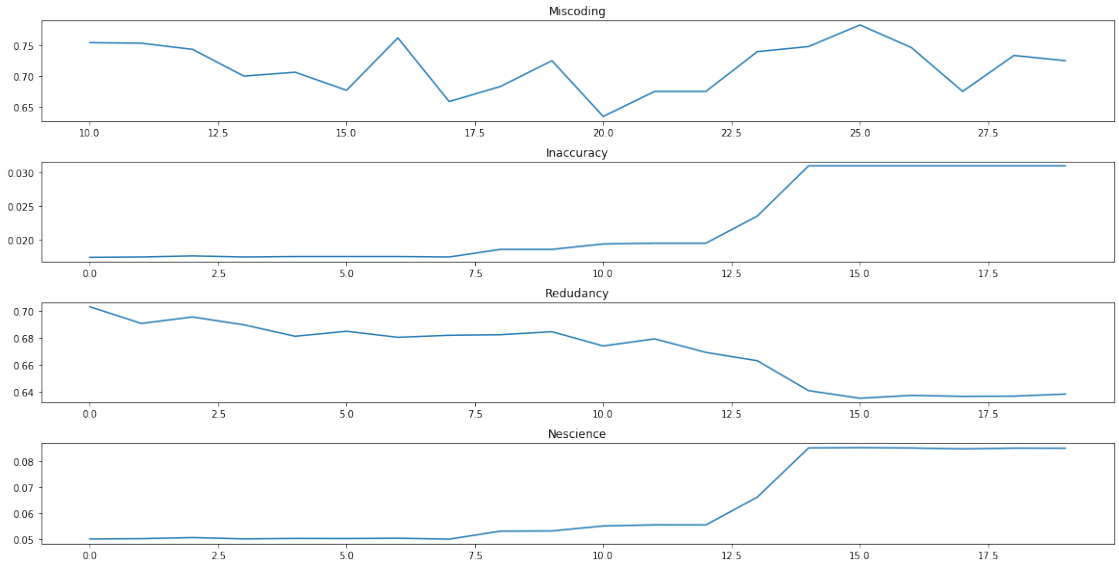
```
Computing optimal codes ... done!
Computing miscoding ... done!
```

[54]:
```
lmiscoding  = list()
linaccuracy = list()
lredudancy  = list()
lnescience  = list()

for i in range(10, 30):

    tree = DecisionTreeClassifier(min_samples_leaf=i)
    tree.fit(X, y)

    attr_in_use = np.zeros(X.shape[1], dtype=int)
    features = set(tree.tree_.feature[tree.tree_.feature >= 0])
    for i in features:
        attr_in_use[i] = 1

    errors = list()
    pred = tree.predict(X)
    for i in np.arange(X.shape[0]):
        if pred[i] != y[i]:
            new_error = list(X[i][np.where(attr_in_use)])
            new_error.append(y[i])
```

```
            errors.append(new_error)
    errors = np.array(errors)

    model_str = tree2string(tree)

    lmiscoding.append(nescience.miscoding(attr_in_use))
    linaccuracy.append(nescience.inaccuracy(errors, attr_in_use))
    lredudancy.append(nescience.redundancy(model_str))

    lnescience.append(nescience.nescience(attr_in_use, errors, model_str))
```

[55]:
```
fig, axs = plt.subplots(4, gridspec_kw={'hspace': 0.4, 'wspace': 0})
axs[0].plot(np.arange(10, 30), lmiscoding)
axs[0].set_title('Miscoding')
axs[1].plot(linaccuracy)
axs[1].set_title('Inaccuracy')
axs[2].plot(lredudancy)
axs[2].set_title('Redudancy')
axs[3].plot(lnescience)
axs[3].set_title('Nescience')
```

[55]:
```
Text(0.5,1,'Nescience')
```



The minimum nescience achieved is

[56]:
```
min(lnescience)
```

[56]:
```
0.049796744658509175
```

And so, the optimal number of samples at leafs should be

[57]:
```
10 + np.where(lnescience == min(lnescience))[0][0]
```

[57]:
```
17
```

Compare the result with the classical way to do this kind of things.

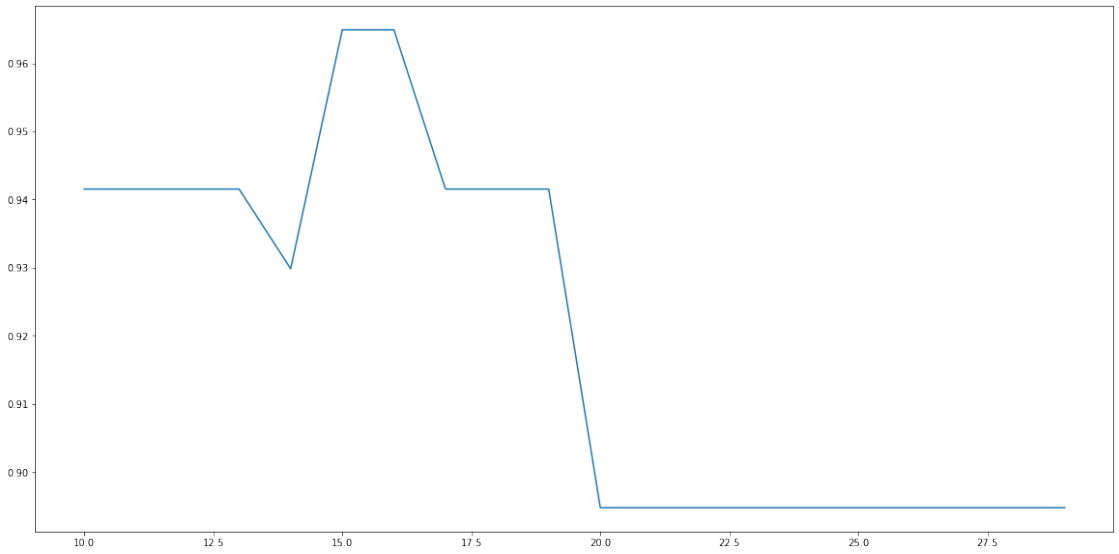
[58]:
```
from sklearn.model_selection import train_test_split
```

[59]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
```

[60]:
```
lscore  = list()

for i in np.arange(10, 30):

    tree = DecisionTreeClassifier(min_samples_leaf=i)
    tree.fit(X_train, y_train)

    score = tree.score(X_test, y_test)

    lscore.append(score)
```

[61]:
```
plt.plot(np.arange(10, 30), lscore)
```

[61]:
```
[<matplotlib.lines.Line2D at 0x7f6a5b0e9278>]
```



[62]:
```
max(lscore)
```

[62]:
```
0.9649122807017544
```

[63]:
```
10 + np.where(lscore == max(lscore))[0][0]
```

[63]:
```
15
```