Promoter Prediction using a Keras Neural Network & Traditional Machine Learning Methods with Scikit-Learn

For gene expression to happen in a cell, proteins (RNA polymerase, sigma factors, etc.) interact with portions of DNA sequence to initiate and otherwise control various aspects of this process. These stretches of DNA are promoters, and in bacterial systems such as *E. coli*, promoters are marked by the presence of -35 and -10 elements, defined as DNA hexamers that are located -35 and -10 nucleotides, respectively, upstream from the transcriptional start site[1].

The following short project evalutes techniques to predict whether a given nucleotide position and its identity are predictive of the sequence belonging to a bacterial promoter or not. The data set comes from the University of California, Irvine, Machine Learning Repository[2], with development described in a separate report[3].

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.callbacks import EarlyStopping

import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import (ExtraTreesClassifier, RandomForestClassifi
                              AdaBoostClassifier, GradientBoostingClassi
from sklearn.svm import SVC
from xgboost import (XGBClassifier, plot_importance, DMatrix)
from sklearn import svm
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import SGDClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import GridSearchCV

from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, roc_auc_score

import shap
import lime
import lime.lime_tabular
```

```
Using TensorFlow backend.
```

In [2]:
```
import warnings
warnings.simplefilter('ignore')
```

In [3]:
```
pro_df = pd.read_csv("promoters.data.txt", delimiter=',', header=None)
pro_df.columns = ['status', 'name', 'seq']
pro_df.head()
```

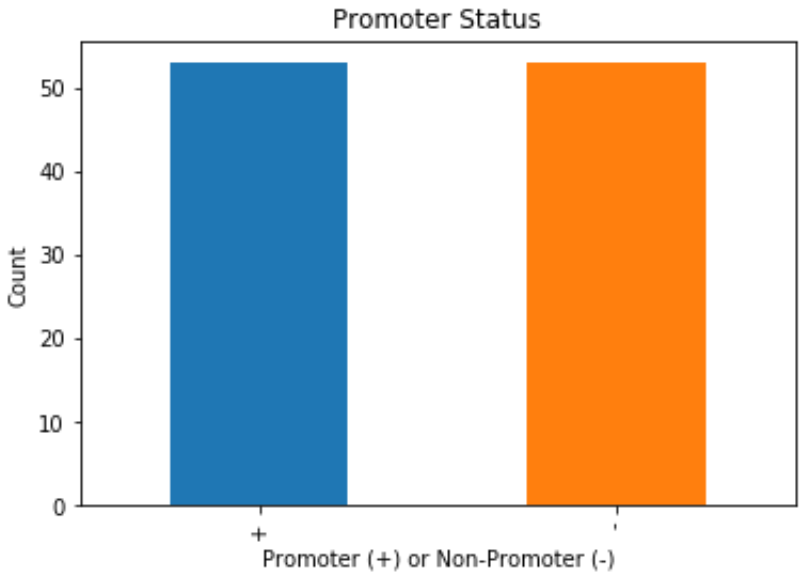
Out[3]:

	status	name	seq
0	+	S10	\t\ttactagcaatacgcttgcgttcggtggttaagtatgtataat...
1	+	AMPC	\t\ttgctatcctgacagttgtcacgctgattggtgtcgttacaat...
2	+	AROH	\t\tgtactagagaactagtgcattagcttatttttttgttatcat...
3	+	DEOP2	\taattgtgatgtgtatcgaagtgtgttgcggagtagatgttagaa...
4	+	LEU1_TRNA	\ttcgataattaactattgacgaaaagctgaaaaccactagaatgc...

In [4]:
```
pro_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 106 entries, 0 to 105
Data columns (total 3 columns):
status    106 non-null object
name      106 non-null object
seq       106 non-null object
dtypes: object(3)
memory usage: 2.6+ KB
```

In [5]:
```
pro_df['status'].value_counts().plot(kind='bar')
plt.title('Promoter Status')
plt.xlabel('Promoter (+) or Non-Promoter (-)')
plt.ylabel('Count')
plt.show()
```



The target class, status of being a promoter or non-promoter sequence, is balanced in this data set. The next step is to transform the status label from a symbol into a numeric value, with "1" for a promoter and "0" for a non-promoter.

In [6]:
```
def recode(value):
    if value == '+':
        return 1
    elif value == '-':
        return 0
    else:
        return value
pro_df['status'] = pro_df.status.apply(recode)
```

In [7]:
```
pro_df.head()
```

Out[7]:

	status	name	seq
0	1	S10	\t\ttactagcaatacgcttgcgttcggtggttaagtatgtataat...
1	1	AMPC	\t\ttgctatcctgacagttgtcacgctgattggtgtcgttacaat...
2	1	AROH	\t\tgtactagagaactagtgcattagcttatttttttgttatcat...
3	1	DEOP2	\taattgtgatgtgtatcgaagtgtgttgcggagtagatgttagaa...
4	1	LEU1_TRNA	\ttcgataattaactattgacgaaaagctgaaaaccactagaatgc...

In [8]:
```
pro_df['seq'] = pro_df.seq.str.replace('\W', '')
```

In [9]:
```
pro_df.head()
```

Out[9]:

	status	name	seq
0	1	S10	tactagcaatacgcttgcgttcggtggttaagtatgtataatgcgc...
1	1	AMPC	tgctatcctgacagttgtcacgctgattggtgtcgttacaatctaa...
2	1	AROH	gtactagagaactagtgcattagcttatttttttgttatcatgcta...
3	1	DEOP2	aattgtgatgtgtatcgaagtgtgttgcggagtagatgttagaata...
4	1	LEU1_TRNA	tcgataattaactattgacgaaaagctgaaaaccactagaatgcgc...

In [10]:
```
sequence = pro_df['seq'].str.strip()
```

In [11]:
```
sequence.head()
```

Out[11]:
```
0    tactagcaatacgcttgcgttcggtggttaagtatgtataatgcgc...
1    tgctatcctgacagttgtcacgctgattggtgtcgttacaatctaa...
2    gtactagagaactagtgcattagcttatttttttgttatcatgcta...
3    aattgtgatgtgtatcgaagtgtgttgcggagtagatgttagaata...
4    tcgataattaactattgacgaaaagctgaaaaccactagaatgcgc...
Name: seq, dtype: object
```

Since each sequence entry is an unusably long sequence, next it is divided one nucleotide at a time, with each nucleotide as its own feature and maintaining its position.

In [12]:
```
seqsplit = []
for n in sequence:
    seqsplit.append(list(n))
```

In [13]:
```
print(seqsplit[0:5])
```

```
[['t', 'a', 'c', 't', 'a', 'g', 'c', 'a', 'a', 't', 'a', 'c', 'g', 'c'
, 't', 't', 'g', 'c', 'g', 't', 't', 'c', 'g', 'g', 't', 'g', 'g', 't'
, 't', 'a', 'a', 'g', 't', 'a', 't', 'g', 't', 'a', 't', 'a', 'a', 't'
, 'g', 'c', 'g', 'c', 'g', 'g', 'g', 'c', 't', 't', 'g', 't', 'c', 'g'
, 't'], ['t', 'g', 'c', 't', 'a', 't', 'c', 'c', 't', 'g', 'a', 'c', '
a', 'g', 't', 't', 'g', 't', 'c', 'a', 'c', 'g', 'c', 't', 'g', 'a', '
t', 't', 'g', 'g', 't', 'g', 't', 'c', 'g', 't', 't', 'a', 'c', 'a', '
a', 't', 'c', 't', 'a', 'a', 'c', 'g', 'c', 'a', 't', 'c', 'g', 'c', '
c', 'a', 'a'], ['g', 't', 'a', 'c', 't', 'a', 'g', 'a', 'g', 'a', 'a',
'c', 't', 'a', 'g', 't', 'g', 'c', 'a', 't', 't', 'a', 'g', 'c', 't',
't', 'a', 't', 't', 't', 't', 't', 't', 't', 'g', 't', 't', 'a', 't',
'c', 'a', 't', 'g', 'c', 't', 'a', 'a', 'c', 'c', 'a', 'c', 'c', 'c',
'g', 'g', 'c', 'g'], ['a', 'a', 't', 't', 'g', 't', 'g', 'a', 't', 'g'
, 't', 'g', 't', 'a', 't', 'c', 'g', 'a', 'a', 'g', 't', 'g', 't', 'g'
, 't', 't', 'g', 'c', 'g', 'g', 'a', 'g', 't', 'a', 'g', 'a', 't', 'g'
, 't', 't', 'a', 'g', 'a', 'a', 't', 'a', 'c', 't', 'a', 'a', 'c', 'a'
, 'a', 'a', 'c', 't', 'c'], ['t', 'c', 'g', 'a', 't', 'a', 'a', 't', '
t', 'a', 'a', 'c', 't', 'a', 't', 't', 'g', 'a', 'c', 'g', 'a', 'a', '
a', 'a', 'g', 'c', 't', 'g', 'a', 'a', 'a', 'a', 'c', 'c', 'a', 'c', '
t', 'a', 'g', 'a', 'a', 't', 'g', 'c', 'g', 'c', 'c', 't', 'c', 'c', '
g', 't', 'g', 'g', 't', 'a', 'g']]
```

In [14]:
```
seq_df = pd.DataFrame(seqsplit, columns = [str(i) for i in range(0,57)])
seq_df.head()
```

Out[14]:

	0	1	2	3	4	5	6	7	8	9	...	47	48	49	50	51	52	53	54	55	56
0	t	a	c	t	a	g	c	a	a	t	...	g	g	c	t	t	g	t	c	g	t
1	t	g	c	t	a	t	c	c	t	g	...	g	c	a	t	c	g	c	c	a	a
2	g	t	a	c	t	a	g	a	g	a	...	c	c	a	c	c	c	g	g	c	g
3	a	a	t	t	g	t	g	a	t	g	...	t	a	a	c	a	a	a	c	t	c
4	t	c	g	a	t	a	a	t	t	a	...	t	c	c	g	t	g	g	t	a	g

5 rows × 57 columns

In [15]:
```
pdf = pd.concat([pro_df, seq_df], axis=1)
pdf.set_index('name', inplace=True)
```

In [16]:
```
pdf = pdf.drop('seq', axis=1)
pdf.head()
```

Out[16]:

	status	0	1	2	3	4	5	6	7	8	...	47	48	49	50	51	52	53	54	55	56
name																					
S10	1	t	a	c	t	a	g	c	a	a	...	g	g	c	t	t	g	t	c	g	t
AMPC	1	t	g	c	t	a	t	c	c	t	...	g	c	a	t	c	g	c	c	a	a
AROH	1	g	t	a	c	t	a	g	a	g	...	c	c	a	c	c	c	g	g	c	g
DEOP2	1	a	a	t	t	g	t	g	a	t	...	t	a	a	c	a	a	a	c	t	c
LEU1_TRNA	1	t	c	g	a	t	a	a	t	t	...	t	c	c	g	t	g	g	t	a	g

5 rows × 58 columns

Predictive analysis

Next, training and test sets are derived and with promoter status (the target) separated from features. The positions of each nucleotide are turned into dummy variables representing each position and its identity. Because of the highly interactive nature of these nucleotide features in their influence on promoter status, the first modeling strategy will use a

In [17]:
```
X = pdf.drop('status', axis = 1)
y = pdf.status
```

In [18]:
```
X = pd.get_dummies(X)
```

In [19]:
```
X.head()
```

Out[19]:

name	0_a	0_c	0_g	0_t	1_a	1_c	1_g	1_t	2_a	2_c	...	54_g	54_t	55_a	55_c	55_g
S10	0	0	0	1	1	0	0	0	0	1	...	0	0	0	0	1
AMPC	0	0	0	1	0	0	1	0	0	1	...	0	0	1	0	0
AROH	0	0	1	0	0	0	0	1	1	0	...	1	0	0	1	0
DEOP2	1	0	0	0	1	0	0	0	0	0	...	0	0	0	0	0
LEU1_TRNA	0	0	0	1	0	1	0	0	0	0	...	0	1	1	0	0

5 rows × 228 columns

In [20]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, ra
```

In [21]:
```
model = Sequential()
model.add(Dense(100, input_dim=228, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(100, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(100, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(100, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(100, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

In [22]:
```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['ac
```

In [23]:
```
early_stopping = EarlyStopping(patience=3)
model.fit(X_train, y_train, epochs=10, validation_split=0.3, callbacks =
```

```
Train on 55 samples, validate on 24 samples
Epoch 1/10
55/55 [==============================] - 1s 16ms/step - loss: 0.6994 -
acc: 0.4909 - val_loss: 0.7057 - val_acc: 0.4167
Epoch 2/10
55/55 [==============================] - 0s 186us/step - loss: 0.6505
- acc: 0.6000 - val_loss: 0.6939 - val_acc: 0.5000
Epoch 3/10
55/55 [==============================] - 0s 228us/step - loss: 0.6459
- acc: 0.6182 - val_loss: 0.6770 - val_acc: 0.5417
Epoch 4/10
55/55 [==============================] - 0s 237us/step - loss: 0.6428
- acc: 0.6727 - val_loss: 0.6528 - val_acc: 0.5833
Epoch 5/10
55/55 [==============================] - 0s 243us/step - loss: 0.5723
- acc: 0.7455 - val_loss: 0.6233 - val_acc: 0.6667
Epoch 6/10
55/55 [==============================] - 0s 271us/step - loss: 0.5423
- acc: 0.8000 - val_loss: 0.5938 - val_acc: 0.6667
Epoch 7/10
55/55 [==============================] - 0s 246us/step - loss: 0.3983
- acc: 0.9455 - val_loss: 0.5646 - val_acc: 0.7083
Epoch 8/10
55/55 [==============================] - 0s 224us/step - loss: 0.3287
- acc: 0.9273 - val_loss: 0.5515 - val_acc: 0.7083
Epoch 9/10
55/55 [==============================] - 0s 255us/step - loss: 0.2476
- acc: 0.9455 - val_loss: 0.5856 - val_acc: 0.7500
Epoch 10/10
55/55 [==============================] - 0s 249us/step - loss: 0.1571
- acc: 0.9818 - val_loss: 0.6081 - val_acc: 0.7917
```

Out[23]:
```
<keras.callbacks.History at 0x1a32369e80>
```

In [24]:
```
scores = model.evaluate(X_train, y_train)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
79/79 [==============================] - 0s 114us/step

acc: 93.67%
```

In [25]:
```
scores1 = model.evaluate(X_test, y_test)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores1[1]*100))
```

```
27/27 [==============================] - 0s 67us/step

acc: 81.48%
```

In [26]:
```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(79, 228)
(27, 228)
(79,)
(27,)
```

In [27]:
```
predictions = model.predict(X_test)
y_pred = predictions[:,0]
print(y_pred[0])
print(y_test[0])
print(y_pred[13])
print(y_test[13])
print(y_pred[3])
print(y_test[3])
```

```
0.97216964
1
0.5286071
1
0.3414873
0
```

Traditional machine learning methods

In [28]:
```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
print('Basic Random Forest accuracy on training data:',round(rf.score(X_
print('Basic Random Forest accuracy on test data:',round(rf.score(X_test
```

```
Basic Random Forest accuracy on training data: 1.0
Basic Random Forest accuracy on test data: 0.815
```

In [29]:
```
rf_pred = rf.predict(X_test)
print(classification_report(y_test,rf_pred))
```

```
              precision    recall  f1-score   support

           0       0.90      0.69      0.78        13
           1       0.76      0.93      0.84        14

   micro avg       0.81      0.81      0.81        27
   macro avg       0.83      0.81      0.81        27
weighted avg       0.83      0.81      0.81        27
```

In [30]:
```
param_grid = {'n_estimators' : [1,3,9,18,36,96,200,600], 'max_depth': [1
CV_rf = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5)
CV_rf.fit(X_train, y_train)
print(CV_rf.best_params_)
print(CV_rf.best_score_)
print(CV_rf.best_estimator_)
```

```
{'max_depth': 3, 'min_samples_leaf': 10, 'n_estimators': 200}
0.9493670886075949
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='g
ini',
            max_depth=3, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=10, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=Non
e,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [31]:
```
rf_tuned = CV_rf.best_estimator_
rf_tuned.fit(X_train, y_train)
y_true, y_predict = y_test, rf_tuned.predict(X_test)
print('Tuned random forest accuracy on training data:',round(rf_tuned.sc
print('Tuned random forest accuracy on test data:',round(rf_tuned.score(
print('\nClassification report:\n',classification_report(y_true,y_predic
print('Confusion matrix:\n',confusion_matrix(y_true,y_predict))
```

```
Tuned random forest accuracy on training data: 0.987
Tuned random forest accuracy on test data: 0.963

Classification report:
               precision    recall  f1-score   support

           0       1.00      0.92      0.96        13
           1       0.93      1.00      0.97        14

   micro avg       0.96      0.96      0.96        27
   macro avg       0.97      0.96      0.96        27
weighted avg       0.97      0.96      0.96        27

Confusion matrix:
 [[12  1]
 [ 0 14]]
```

In [32]:
```
mlp = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
mlp.fit(X_train, y_train)
print('Accuracy on training data:',round(mlp.score(X_train, y_train),3))
print('Accuracy on test data:',round(mlp.score(X_test, y_test),3))

predmlp = mlp.predict(X_test)
print(classification_report(y_test,predmlp))
```

```
Accuracy on training data: 1.0
Accuracy on test data: 0.926
              precision    recall  f1-score   support

           0       1.00      0.85      0.92        13
           1       0.88      1.00      0.93        14

   micro avg       0.93      0.93      0.93        27
   macro avg       0.94      0.92      0.93        27
weighted avg       0.94      0.93      0.93        27
```

In [33]:
```
xgb = XGBClassifier()
xgb.fit(X_train, y_train)
y_pred = xgb.predict(X_test)
predictions1 = [round(value) for value in y_pred]
print('Accuracy on training data:',round(xgb.score(X_train, y_train),3))
print('Accuracy on test data:',round(xgb.score(X_test, y_test),3))
print(classification_report(y_test,predictions1))
```

```
Accuracy on training data: 1.0
Accuracy on test data: 0.815
              precision    recall  f1-score   support

           0       0.90      0.69      0.78        13
           1       0.76      0.93      0.84        14

   micro avg       0.81      0.81      0.81        27
   macro avg       0.83      0.81      0.81        27
weighted avg       0.83      0.81      0.81        27
```

In [34]:
```
logr = LogisticRegression(class_weight='balanced')
logr.fit(X_train, y_train)
ly_pred = logr.predict(X_test)
print('Accuracy on training data:',round(logr.score(X_train, y_train),3))
print('Accuracy on test data:',round(logr.score(X_test, y_test),3))
print(classification_report(y_test,ly_pred))
print(confusion_matrix(y_test,ly_pred))
```

```
Accuracy on training data: 1.0
Accuracy on test data: 0.926
              precision    recall  f1-score   support

           0       1.00      0.85      0.92        13
           1       0.88      1.00      0.93        14

   micro avg       0.93      0.93      0.93        27
   macro avg       0.94      0.92      0.93        27
weighted avg       0.94      0.93      0.93        27

[[11  2]
 [ 0 14]]
```

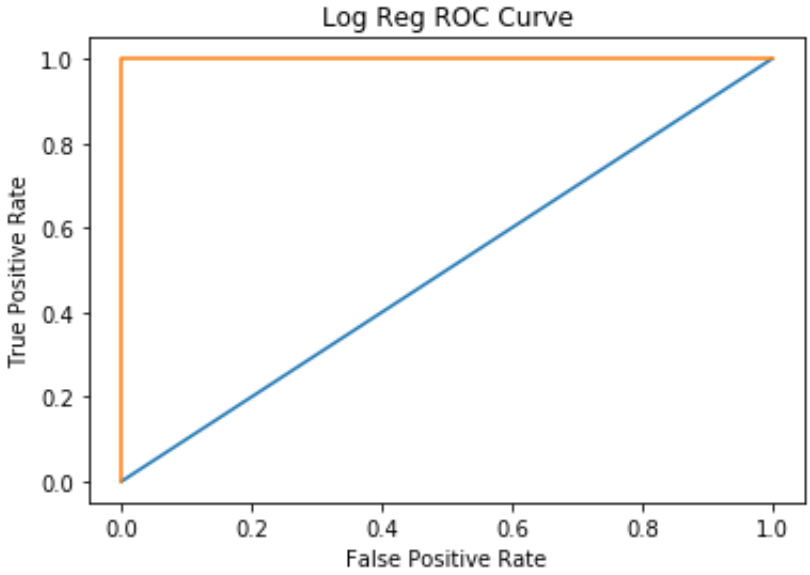
In [35]:
```
mnb = MultinomialNB()
mnb.fit(X_train, y_train)
mny_pred = mnb.predict(X_test)
print('Accuracy on training data:',round(mnb.score(X_train, y_train),3))
print('Accuracy on test data:',round(mnb.score(X_test, y_test),3))
print(classification_report(y_test,mny_pred))
print(confusion_matrix(y_test,mny_pred))
```

```
Accuracy on training data: 1.0
Accuracy on test data: 0.926
              precision    recall  f1-score   support

           0       1.00      0.85      0.92        13
           1       0.88      1.00      0.93        14

   micro avg       0.93      0.93      0.93        27
   macro avg       0.94      0.92      0.93        27
weighted avg       0.94      0.93      0.93        27

[[11  2]
 [ 0 14]]
```

AUC-ROC for Scikit-Learn logistic regression model

In [36]:
```
ly_pred_prob = logr.predict_proba(X_test)[:,1]
print('AUC-ROC score for tuned log reg model:', round(roc_auc_score(y_te
fpr, tpr, thresholds = roc_curve(y_test, ly_pred_prob)
plt.plot([0,1],[0,1])
plt.plot(fpr,tpr, label='LR')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Log Reg ROC Curve')
plt.show()
```

```
AUC-ROC score for tuned log reg model: 1.0
```



In [37]:
```
lr_pred = logr.predict(X_test)
print(list(zip(y_test,lr_pred)))
print(confusion_matrix(y_test, lr_pred))
```

```
[(1, 1), (0, 0), (0, 0), (0, 0), (1, 1), (0, 0), (0, 1), (1, 1), (0, 0
), (1, 1), (1, 1), (0, 0), (1, 1), (1, 1), (1, 1), (1, 1), (0, 0), (1,
1), (0, 1), (0, 0), (1, 1), (1, 1), (0, 0), (0, 0), (1, 1), (0, 0), (1
, 1)]
[[11  2]
 [ 0 14]]
```

The logistic regression model shown here seems to be the best or tied for the best among the tested models here. Based on accuracy scores, and the AUC for the ROC plot for the logistic regression model, it is a highly effective model for this analysis of predicting which DNA sequences are promoters or not, and also is a computationally straightforward method.

Model interpretability is useful to explore if interested in knowing which features (in this case nucleotide positions and identities) contribute to predictions and in which manner. For an analysis such as this one, though a model interpretability analysis may appear to suggest otherwise, it should not be expected that each nucleotide position is operating independently; adjacent nucleotides within a promoter element, as well as other possible elements, necessarily function together in interactions with DNA-binding proteins.

SHAP results are presented with a few individual predictions shown and how key nucleotide positions and identities contributed to them. Positive predictions for promoter status point to the right (closer to 1), while sequences predicted to be non-promoters point to the left. Following these, a summary plot is shown, depicting which positions and identities were most influential. After these are LIME interpretability analyses[5] for the same logistic regression model as that used in SHAP analysis (for one individual sample) and then for the Keras model. Note: individual prediction images are not displayed on GitHub, but screenshots are available in the GitHub folder containing this notebook. Screenshots of LIME output include only a portion of results.

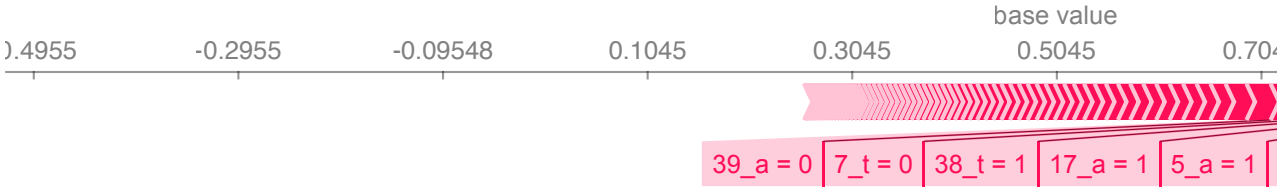
SHAP with Scikit-Learn logistic regression model

In [38]:
```
shap.initjs()

explainer = shap.KernelExplainer(logr.predict_proba, X_train)

shap_values = explainer.shap_values(X_train)
```

```
100%|██████████| 79/79 [02:18<00:00,  1.78s/it]
```

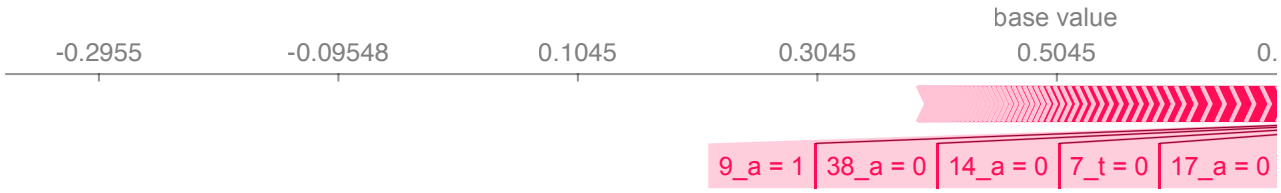
In [39]:
```
shap.force_plot(explainer.expected_value[0], shap_values[1][0,:], X_test
```

Out[39]:



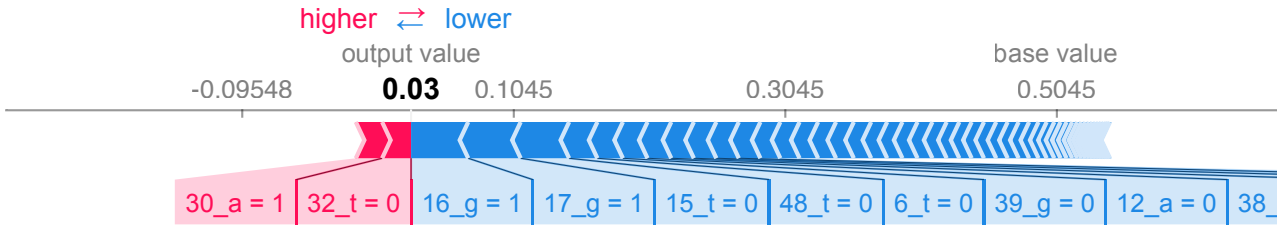
In [40]:
```
shap.force_plot(explainer.expected_value[0], shap_values[1][13,:], X_tes
```

Out[40]:

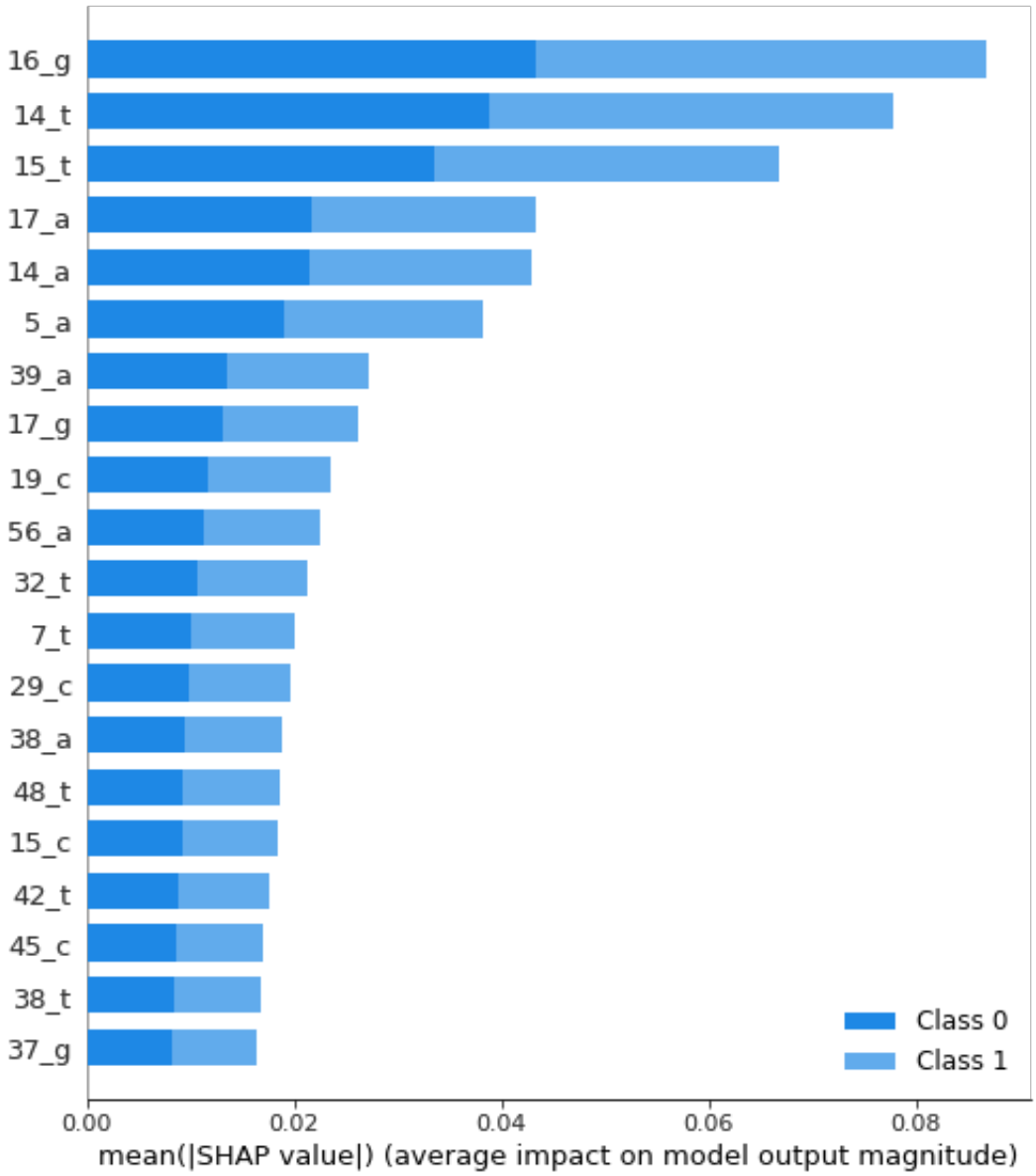


In [41]:
```
shap.force_plot(explainer.expected_value[0], shap_values[1][3,:], X_test
```

Out[41]:



In [42]:
```
shap.summary_plot(shap_values, X_test)
```



While the documentation with the SHAP package explains that it unifies multiple methods[4], including the LIME model explainer, using the LIME package itself provides an expanded view of the predictive contributions of all features (228 here), which may be useful for some purposes. The two packages may overlap but are not identical in approach, so different prediction probabilities can arise.

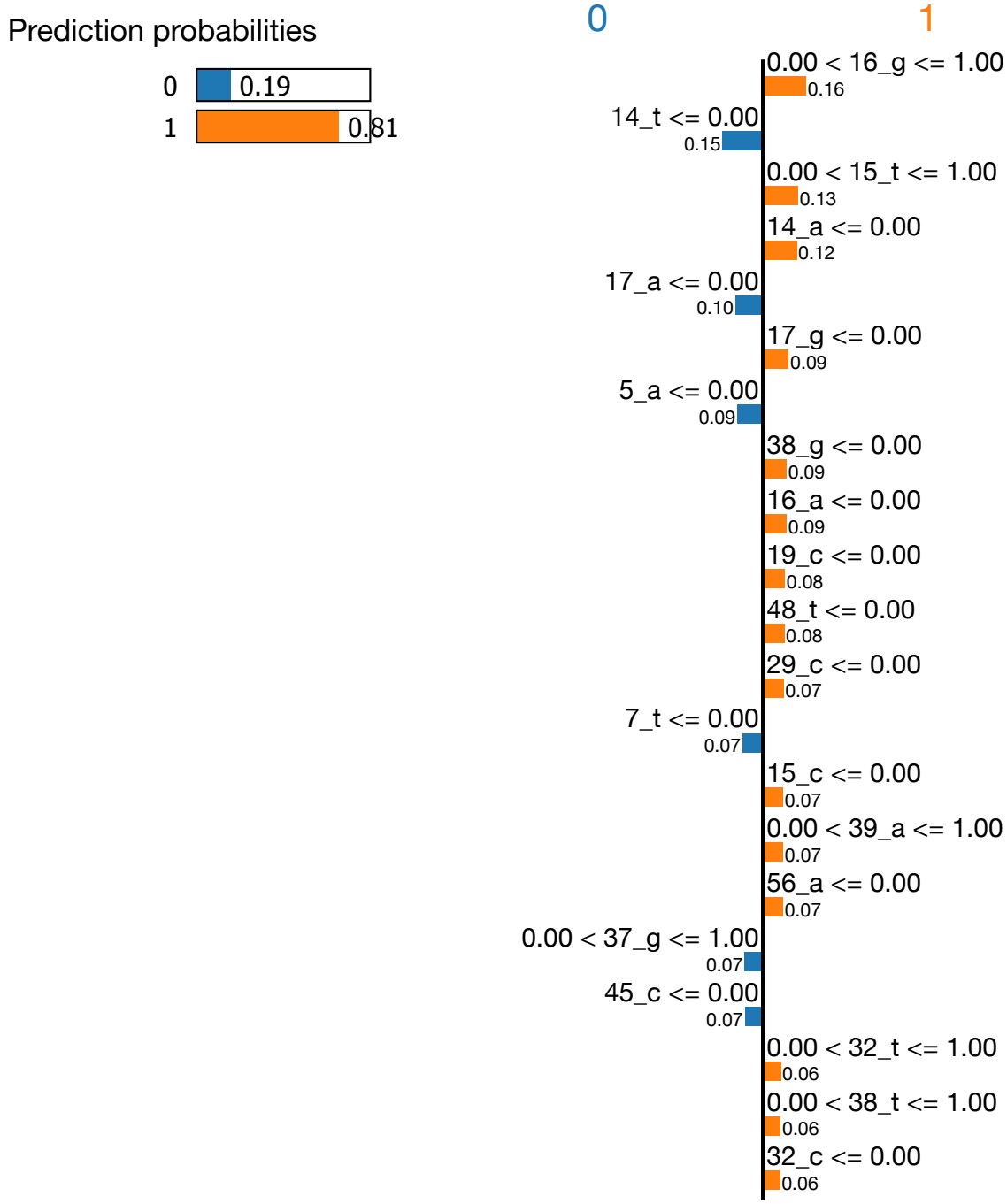
With the Keras neural network model, LIME produces an error regarding probabilities because the "predict_proba" function works differently with Keras models than with Scikit-Learn. An alternative is to use "predict_classes", but this produces an all-or-nothing result that may obscure differences between samples.

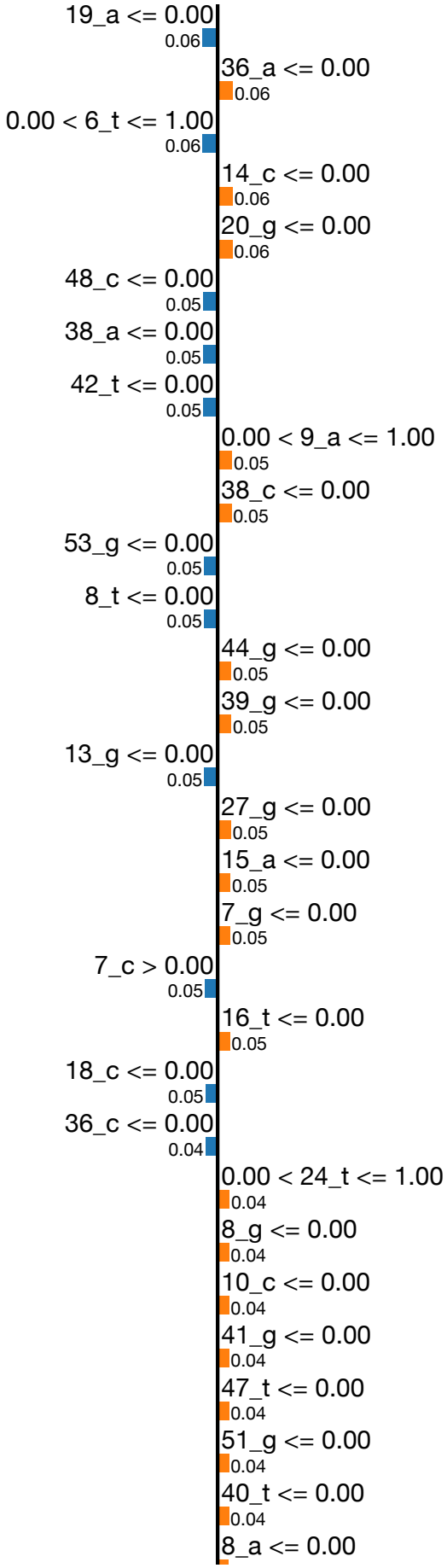
LIME with Scikit-Learn logistic regression model

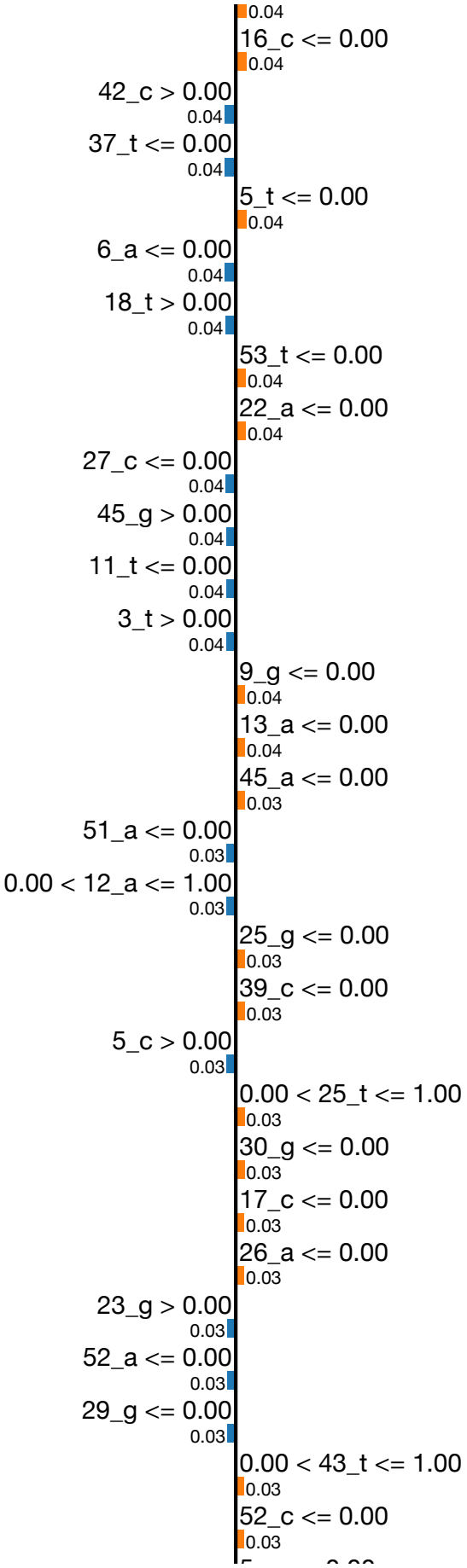
In [43]:
```
explainer = lime.lime_tabular.LimeTabularExplainer(X_train.values, featu
                                                   class_names=[0,1], di
```

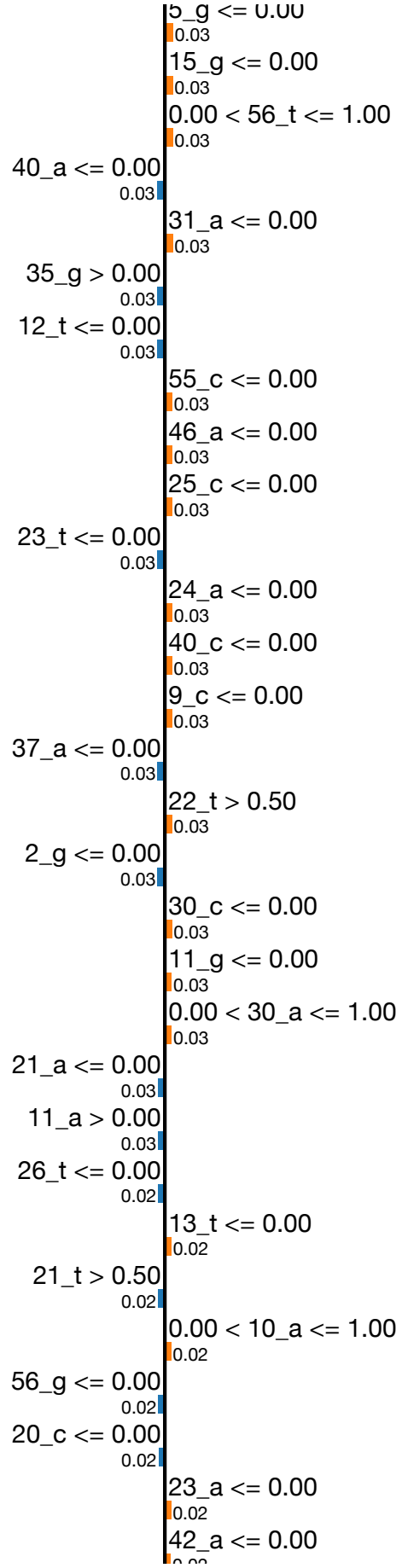
In [44]:
```
i = 13 #sample number from X_test
exp = explainer.explain_instance(X_test.values[i], logr.predict_proba, n
```

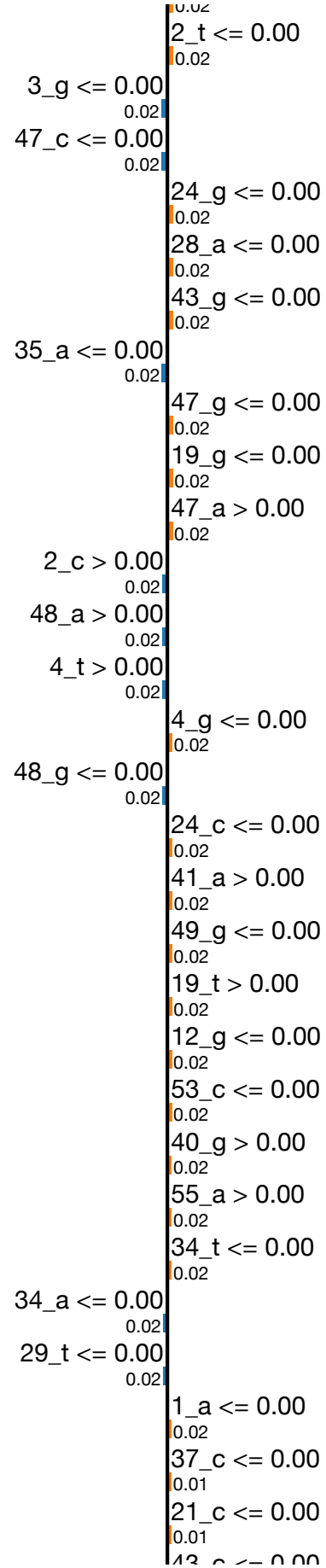
In [45]:
```
exp.show_in_notebook(show_table=True, show_all=False)
```

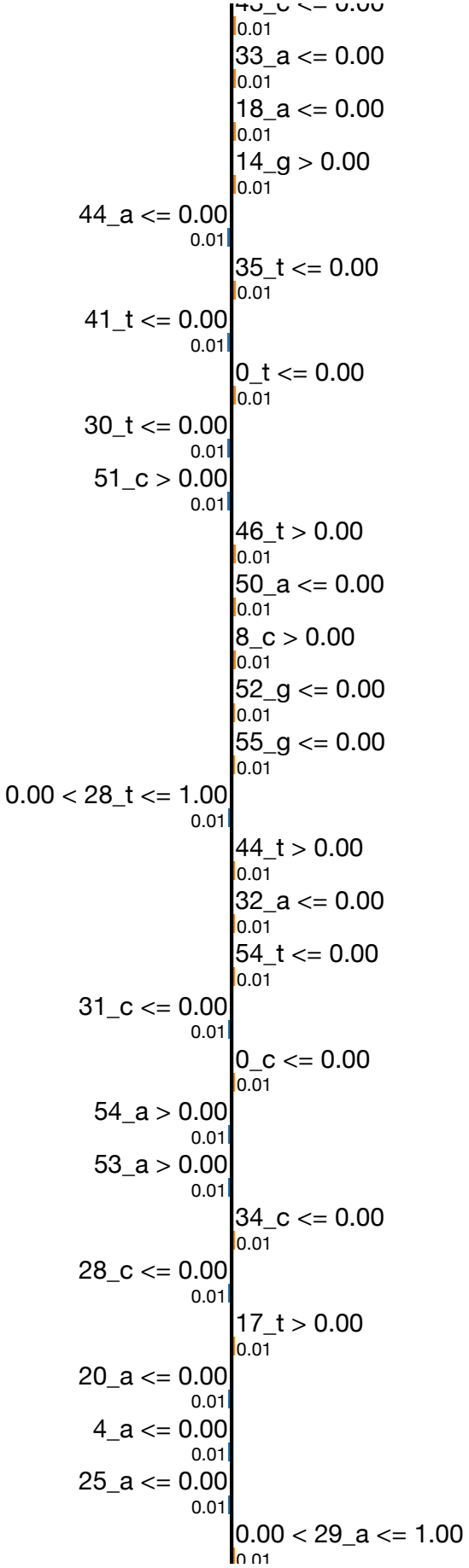


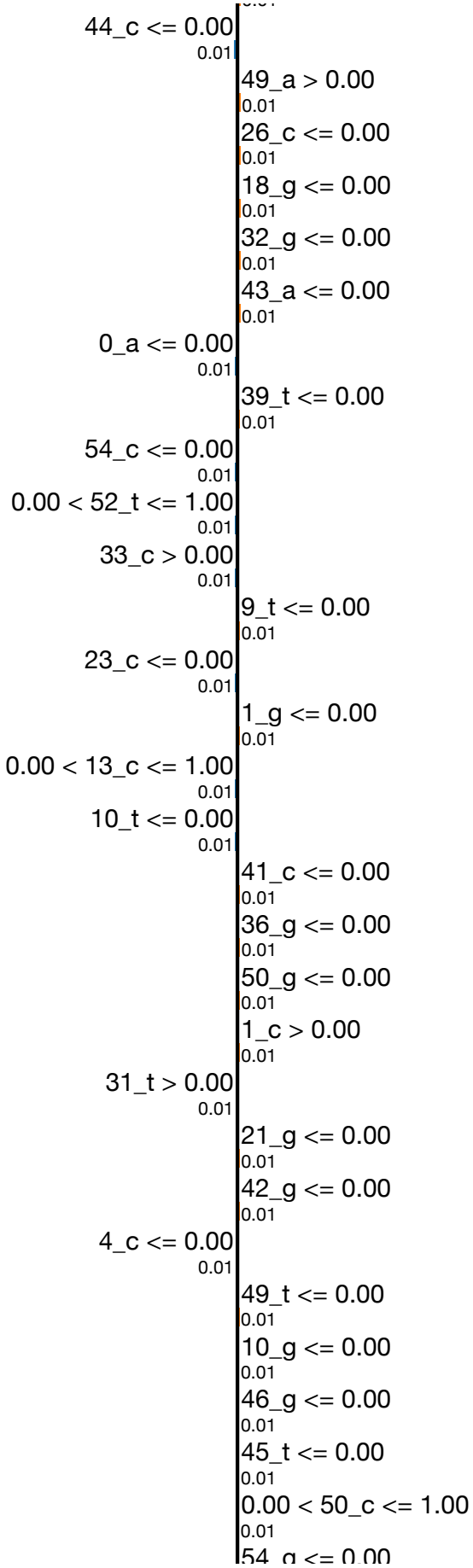

19_a <= 0.00
0.06
36_a <= 0.00
0.06
0.00 < 6_t <= 1.00
0.06
14_c <= 0.00
0.06
20_g <= 0.00
0.06
48_c <= 0.00
0.05
38_a <= 0.00
0.05
42_t <= 0.00
0.05
0.00 < 9_a <= 1.00
0.05
38_c <= 0.00
0.05
53_g <= 0.00
0.05
8_t <= 0.00
0.05
44_g <= 0.00
0.05
39_g <= 0.00
0.05
13_g <= 0.00
0.05
27_g <= 0.00
0.05
15_a <= 0.00
0.05
7_g <= 0.00
0.05
7_c > 0.00
0.05
16_t <= 0.00
0.05
18_c <= 0.00
0.05
36_c <= 0.00
0.04
0.00 < 24_t <= 1.00
0.04
8_g <= 0.00
0.04
10_c <= 0.00
0.04
41_g <= 0.00
0.04
47_t <= 0.00
0.04
51_g <= 0.00
0.04
40_t <= 0.00
0.04
8_a <= 0.00

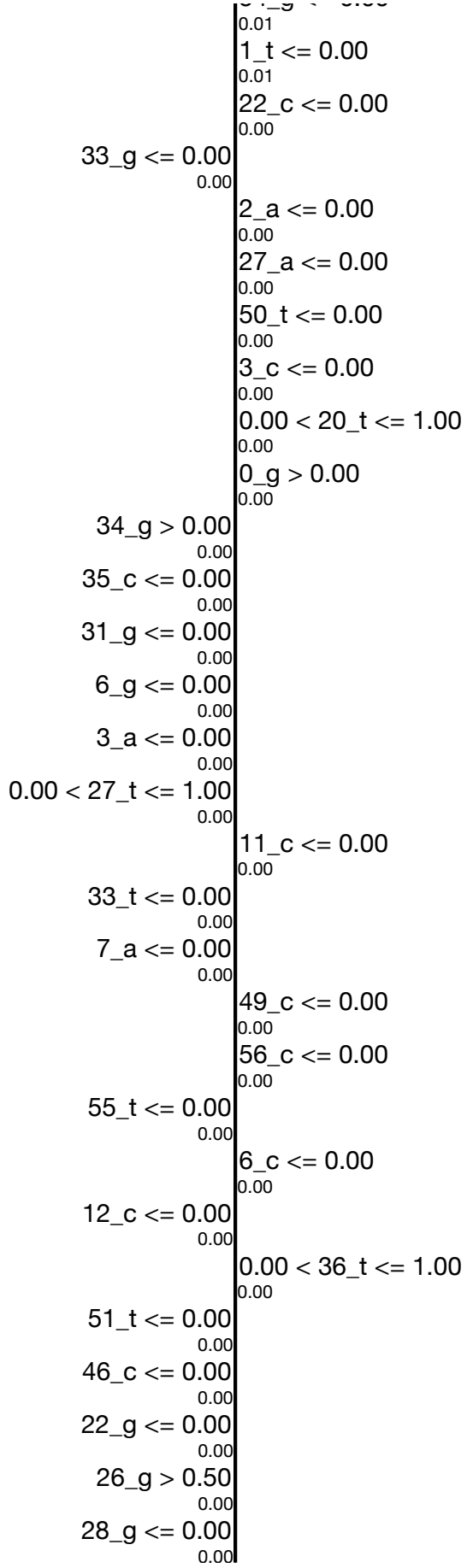

0.04
16_c <= 0.00
0.04
42_c > 0.00
0.04
37_t <= 0.00
0.04
5_t <= 0.00
0.04
6_a <= 0.00
0.04
18_t > 0.00
0.04
53_t <= 0.00
0.04
22_a <= 0.00
0.04
27_c <= 0.00
0.04
45_g > 0.00
0.04
11_t <= 0.00
0.04
3_t > 0.00
0.04
9_g <= 0.00
0.04
13_a <= 0.00
0.04
45_a <= 0.00
0.03
51_a <= 0.00
0.03
0.00 < 12_a <= 1.00
0.03
25_g <= 0.00
0.03
39_c <= 0.00
0.03
5_c > 0.00
0.03
0.00 < 25_t <= 1.00
0.03
30_g <= 0.00
0.03
17_c <= 0.00
0.03
26_a <= 0.00
0.03
23_g > 0.00
0.03
52_a <= 0.00
0.03
29_g <= 0.00
0.03
0.00 < 43_t <= 1.00
0.03
52_c <= 0.00
0.03


5_g <= 0.00
0.03
15_g <= 0.00
0.03
0.00 < 56_t <= 1.00
0.03
40_a <= 0.00
0.03
31_a <= 0.00
0.03
35_g > 0.00
0.03
12_t <= 0.00
0.03
55_c <= 0.00
0.03
46_a <= 0.00
0.03
25_c <= 0.00
0.03
23_t <= 0.00
0.03
24_a <= 0.00
0.03
40_c <= 0.00
0.03
9_c <= 0.00
0.03
37_a <= 0.00
0.03
22_t > 0.50
0.03
2_g <= 0.00
0.03
30_c <= 0.00
0.03
11_g <= 0.00
0.03
0.00 < 30_a <= 1.00
0.03
21_a <= 0.00
0.03
11_a > 0.00
0.03
26_t <= 0.00
0.02
13_t <= 0.00
0.02
21_t > 0.50
0.02
0.00 < 10_a <= 1.00
0.02
56_g <= 0.00
0.02
20_c <= 0.00
0.02
23_a <= 0.00
0.02
42_a <= 0.00


0.02
2_t <= 0.00
0.02
3_g <= 0.00
0.02
47_c <= 0.00
0.02
24_g <= 0.00
0.02
28_a <= 0.00
0.02
43_g <= 0.00
0.02
35_a <= 0.00
0.02
47_g <= 0.00
0.02
19_g <= 0.00
0.02
47_a > 0.00
0.02
2_c > 0.00
0.02
48_a > 0.00
0.02
4_t > 0.00
0.02
4_g <= 0.00
0.02
48_g <= 0.00
0.02
24_c <= 0.00
0.02
41_a > 0.00
0.02
49_g <= 0.00
0.02
19_t > 0.00
0.02
12_g <= 0.00
0.02
53_c <= 0.00
0.02
40_g > 0.00
0.02
55_a > 0.00
0.02
34_t <= 0.00
0.02
34_a <= 0.00
0.02
29_t <= 0.00
0.02
1_a <= 0.00
0.02
37_c <= 0.00
0.01
21_c <= 0.00
0.01


43_c <= 0.00
0.01
33_a <= 0.00
0.01
18_a <= 0.00
0.01
14_g > 0.00
0.01
44_a <= 0.00
0.01
35_t <= 0.00
0.01
41_t <= 0.00
0.01
0_t <= 0.00
0.01
30_t <= 0.00
0.01
51_c > 0.00
0.01
46_t > 0.00
0.01
50_a <= 0.00
0.01
8_c > 0.00
0.01
52_g <= 0.00
0.01
55_g <= 0.00
0.01
0.00 < 28_t <= 1.00
0.01
44_t > 0.00
0.01
32_a <= 0.00
0.01
54_t <= 0.00
0.01
31_c <= 0.00
0.01
0_c <= 0.00
0.01
54_a > 0.00
0.01
53_a > 0.00
0.01
34_c <= 0.00
0.01
28_c <= 0.00
0.01
17_t > 0.00
0.01
20_a <= 0.00
0.01
4_a <= 0.00
0.01
25_a <= 0.00
0.01
0.00 < 29_a <= 1.00
0.01


44_c <= 0.00
0.01
49_a > 0.00
0.01
26_c <= 0.00
0.01
18_g <= 0.00
0.01
32_g <= 0.00
0.01
43_a <= 0.00
0.01
0_a <= 0.00
0.01
39_t <= 0.00
0.01
54_c <= 0.00
0.01
0.00 < 52_t <= 1.00
0.01
33_c > 0.00
0.01
9_t <= 0.00
0.01
23_c <= 0.00
0.01
1_g <= 0.00
0.01
0.00 < 13_c <= 1.00
0.01
10_t <= 0.00
0.01
41_c <= 0.00
0.01
36_g <= 0.00
0.01
50_g <= 0.00
0.01
1_c > 0.00
0.01
31_t > 0.00
0.01
21_g <= 0.00
0.01
42_g <= 0.00
0.01
4_c <= 0.00
0.01
49_t <= 0.00
0.01
10_g <= 0.00
0.01
46_g <= 0.00
0.01
45_t <= 0.00
0.01
0.00 < 50_c <= 1.00
0.01

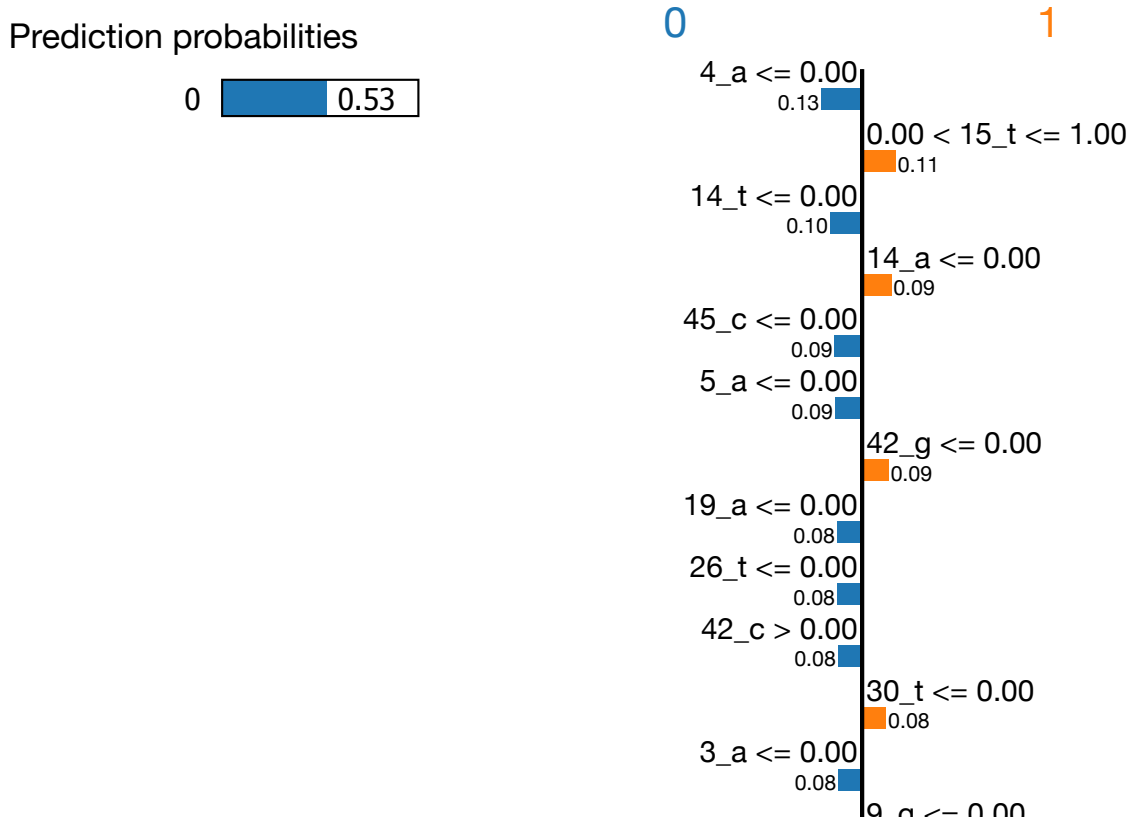

0.01
1_t <= 0.00
0.01
22_c <= 0.00
0.00
33_g <= 0.00
0.00
2_a <= 0.00
0.00
27_a <= 0.00
0.00
50_t <= 0.00
0.00
3_c <= 0.00
0.00
0.00 < 20_t <= 1.00
0.00
0_g > 0.00
0.00
34_g > 0.00
0.00
35_c <= 0.00
0.00
31_g <= 0.00
0.00
6_g <= 0.00
0.00
3_a <= 0.00
0.00
0.00 < 27_t <= 1.00
0.00
11_c <= 0.00
0.00
33_t <= 0.00
0.00
7_a <= 0.00
0.00
49_c <= 0.00
0.00
56_c <= 0.00
0.00
55_t <= 0.00
0.00
6_c <= 0.00
0.00
12_c <= 0.00
0.00
0.00 < 36_t <= 1.00
0.00
51_t <= 0.00
0.00
46_c <= 0.00
0.00
22_g <= 0.00
0.00
26_g > 0.50
0.00
28_g <= 0.00
0.00

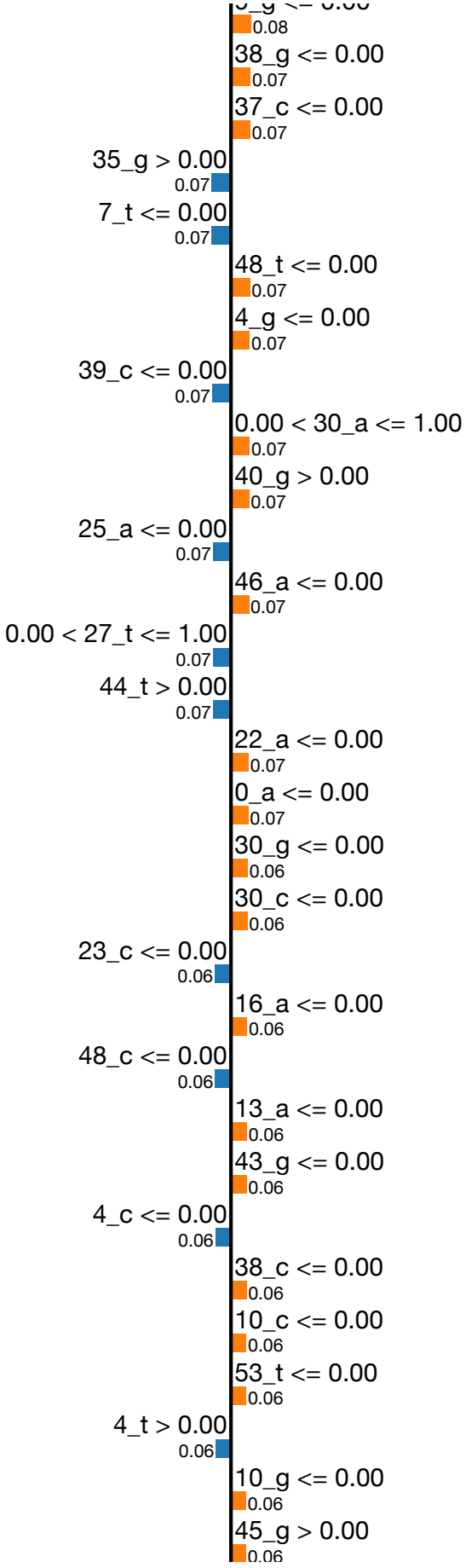
Feature	Value
16_g	1.00
14_t	0.00
15_t	1.00
14_a	0.00
17_a	0.00
17_g	0.00
5_a	0.00
38_g	0.00
16_a	0.00
19_c	0.00

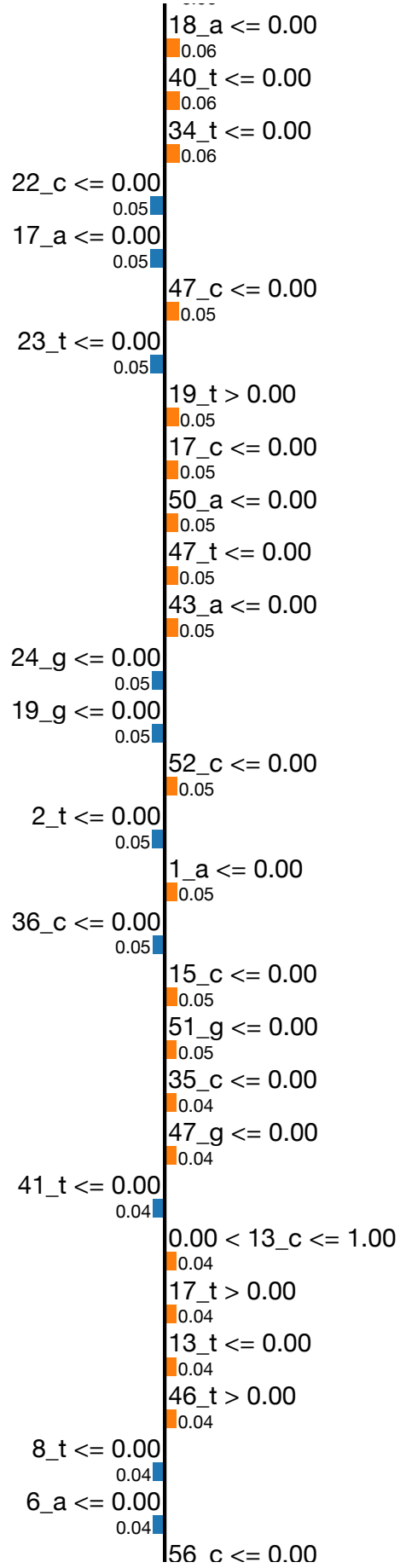
LIME with Keras model

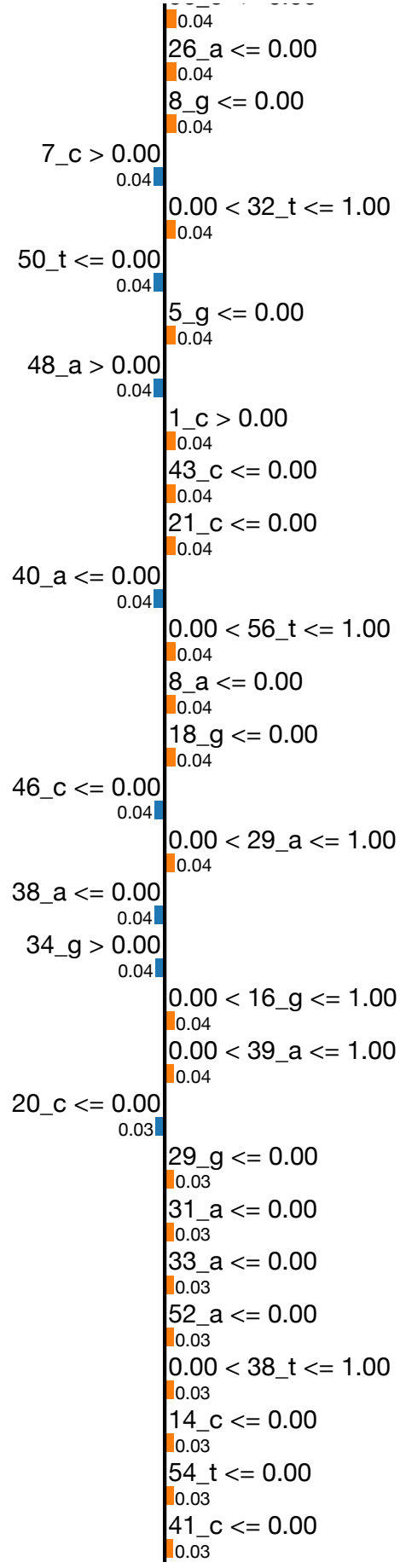
In [46]:
```
j = 13 #sample number from X_test
expnn = explainer.explain_instance(X_test.values[j], model.predict_proba
```

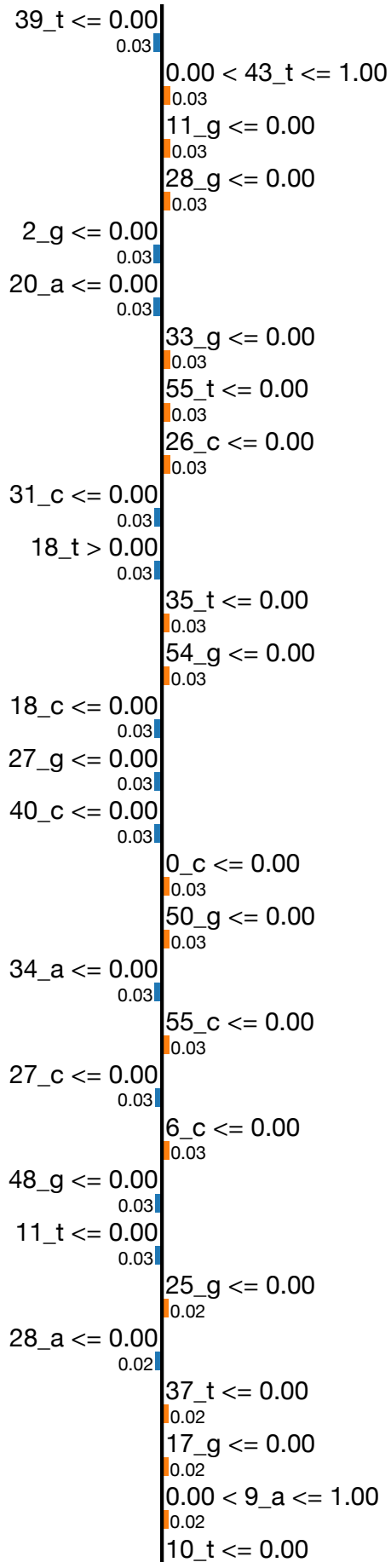
In [47]:
```
expnn.show_in_notebook(show_table=True, show_all=False)
```

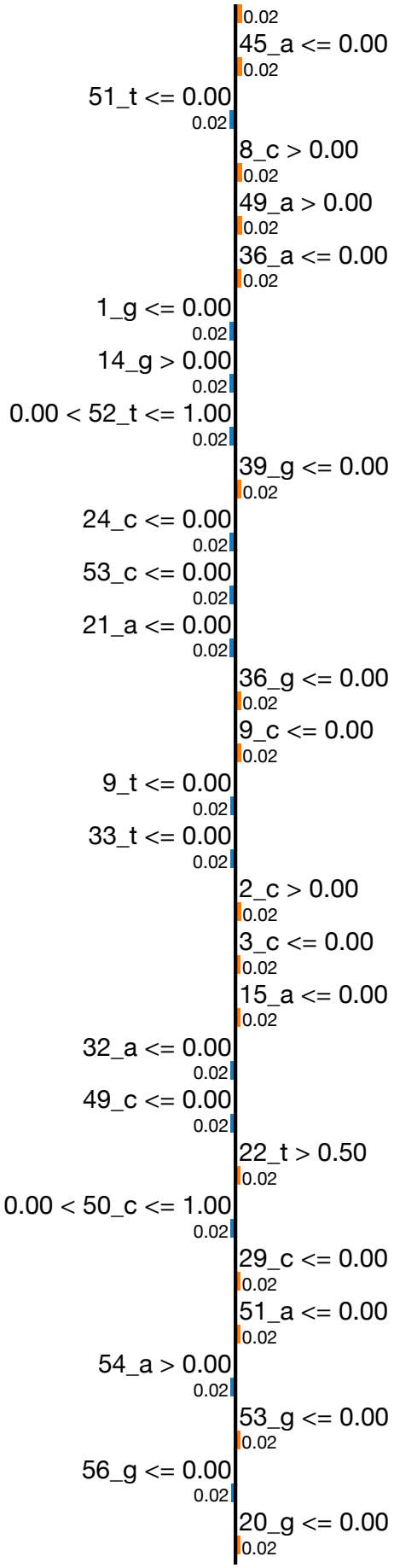


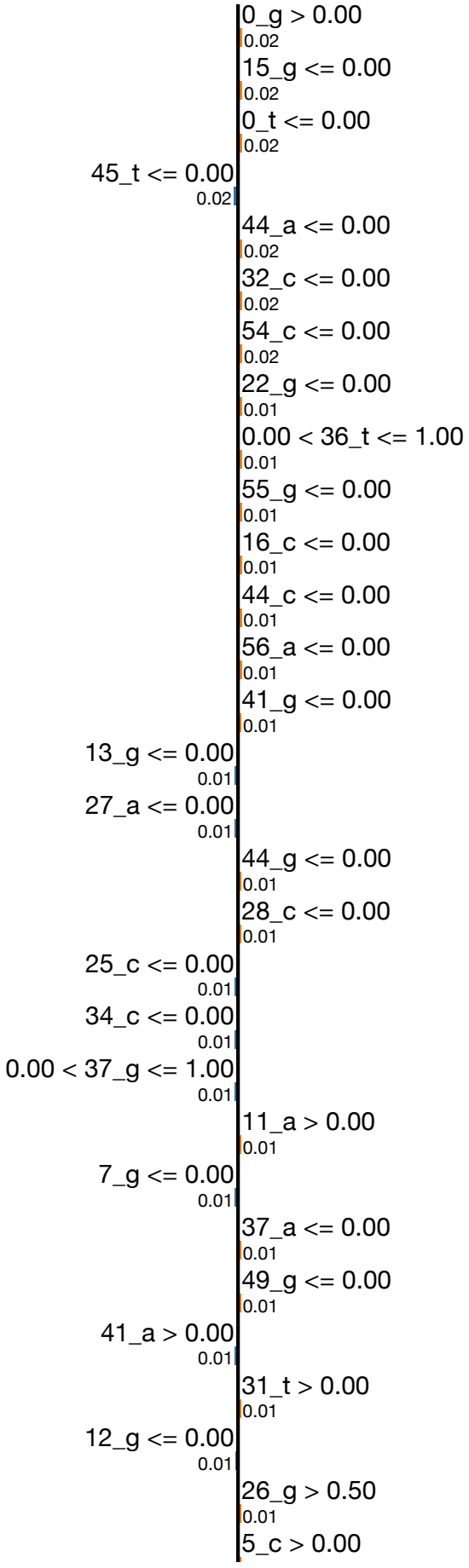

0.08
38_g <= 0.00
0.07
37_c <= 0.00
0.07
35_g > 0.00
0.07
7_t <= 0.00
0.07
48_t <= 0.00
0.07
4_g <= 0.00
0.07
39_c <= 0.00
0.07
0.00 < 30_a <= 1.00
0.07
40_g > 0.00
0.07
25_a <= 0.00
0.07
46_a <= 0.00
0.07
0.00 < 27_t <= 1.00
0.07
44_t > 0.00
0.07
22_a <= 0.00
0.07
0_a <= 0.00
0.07
30_g <= 0.00
0.06
30_c <= 0.00
0.06
23_c <= 0.00
0.06
16_a <= 0.00
0.06
48_c <= 0.00
0.06
13_a <= 0.00
0.06
43_g <= 0.00
0.06
4_c <= 0.00
0.06
38_c <= 0.00
0.06
10_c <= 0.00
0.06
53_t <= 0.00
0.06
4_t > 0.00
0.06
10_g <= 0.00
0.06
45_g > 0.00
0.06

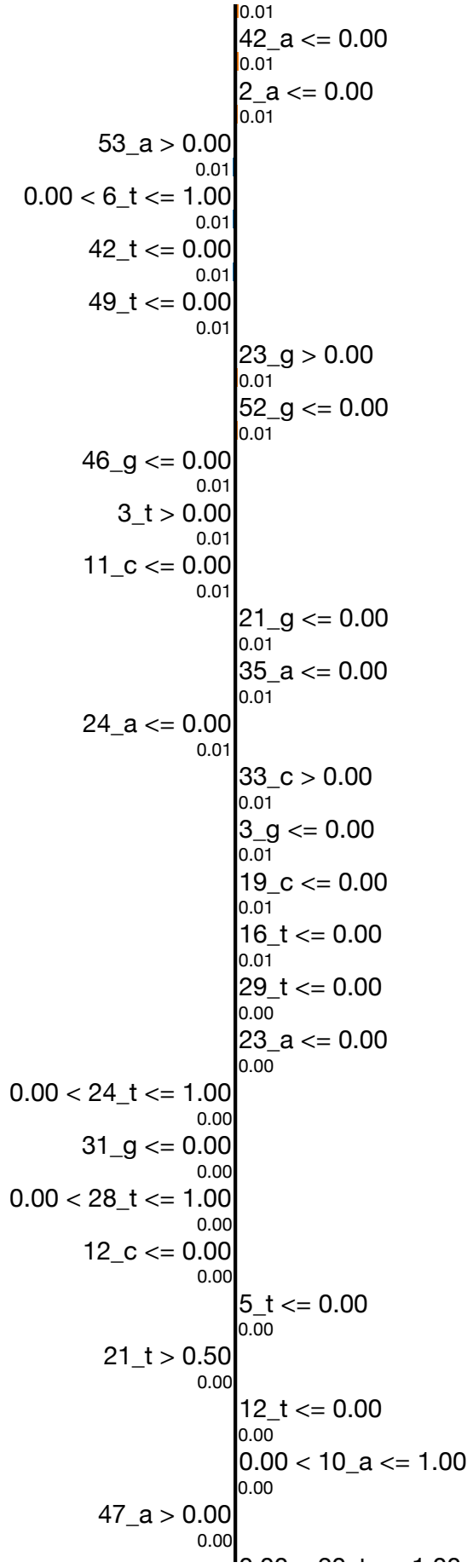

18_a <= 0.00
0.06
40_t <= 0.00
0.06
34_t <= 0.00
0.06
22_c <= 0.00
0.05
17_a <= 0.00
0.05
47_c <= 0.00
0.05
23_t <= 0.00
0.05
19_t > 0.00
0.05
17_c <= 0.00
0.05
50_a <= 0.00
0.05
47_t <= 0.00
0.05
43_a <= 0.00
0.05
24_g <= 0.00
0.05
19_g <= 0.00
0.05
52_c <= 0.00
0.05
2_t <= 0.00
0.05
1_a <= 0.00
0.05
36_c <= 0.00
0.05
15_c <= 0.00
0.05
51_g <= 0.00
0.05
35_c <= 0.00
0.04
47_g <= 0.00
0.04
41_t <= 0.00
0.04
0.00 < 13_c <= 1.00
0.04
17_t > 0.00
0.04
13_t <= 0.00
0.04
46_t > 0.00
0.04
8_t <= 0.00
0.04
6_a <= 0.00
0.04
56_c <= 0.00

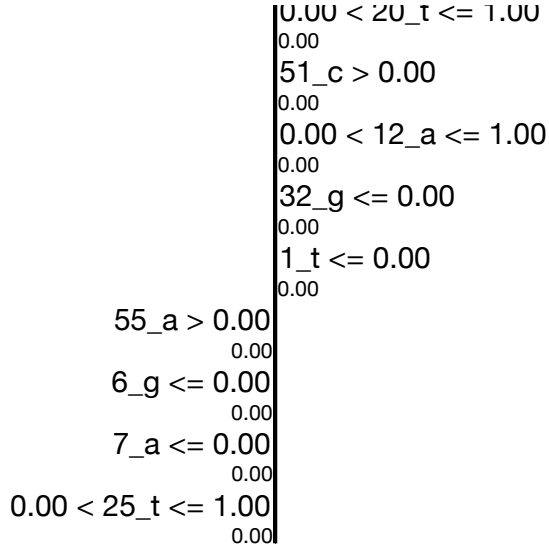

0.04
26_a <= 0.00
0.04
8_g <= 0.00
0.04
7_c > 0.00
0.04
0.00 < 32_t <= 1.00
0.04
50_t <= 0.00
0.04
5_g <= 0.00
0.04
48_a > 0.00
0.04
1_c > 0.00
0.04
43_c <= 0.00
0.04
21_c <= 0.00
0.04
40_a <= 0.00
0.04
0.00 < 56_t <= 1.00
0.04
8_a <= 0.00
0.04
18_g <= 0.00
0.04
46_c <= 0.00
0.04
0.00 < 29_a <= 1.00
0.04
38_a <= 0.00
0.04
34_g > 0.00
0.04
0.00 < 16_g <= 1.00
0.04
0.00 < 39_a <= 1.00
0.04
20_c <= 0.00
0.03
29_g <= 0.00
0.03
31_a <= 0.00
0.03
33_a <= 0.00
0.03
52_a <= 0.00
0.03
0.00 < 38_t <= 1.00
0.03
14_c <= 0.00
0.03
54_t <= 0.00
0.03
41_c <= 0.00
0.03


39_t <= 0.00
0.03
0.00 < 43_t <= 1.00
0.03
11_g <= 0.00
0.03
28_g <= 0.00
0.03
2_g <= 0.00
0.03
20_a <= 0.00
0.03
33_g <= 0.00
0.03
55_t <= 0.00
0.03
26_c <= 0.00
0.03
31_c <= 0.00
0.03
18_t > 0.00
0.03
35_t <= 0.00
0.03
54_g <= 0.00
0.03
18_c <= 0.00
0.03
27_g <= 0.00
0.03
40_c <= 0.00
0.03
0_c <= 0.00
0.03
50_g <= 0.00
0.03
34_a <= 0.00
0.03
55_c <= 0.00
0.03
27_c <= 0.00
0.03
6_c <= 0.00
0.03
48_g <= 0.00
0.03
11_t <= 0.00
0.03
25_g <= 0.00
0.02
28_a <= 0.00
0.02
37_t <= 0.00
0.02
17_g <= 0.00
0.02
0.00 < 9_a <= 1.00
0.02
10_t <= 0.00


0.02
45_a <= 0.00
0.02
51_t <= 0.00
0.02
8_c > 0.00
0.02
49_a > 0.00
0.02
36_a <= 0.00
0.02
1_g <= 0.00
0.02
14_g > 0.00
0.02
0.00 < 52_t <= 1.00
0.02
39_g <= 0.00
0.02
24_c <= 0.00
0.02
53_c <= 0.00
0.02
21_a <= 0.00
0.02
36_g <= 0.00
0.02
9_c <= 0.00
0.02
9_t <= 0.00
0.02
33_t <= 0.00
0.02
2_c > 0.00
0.02
3_c <= 0.00
0.02
15_a <= 0.00
0.02
32_a <= 0.00
0.02
49_c <= 0.00
0.02
22_t > 0.50
0.02
0.00 < 50_c <= 1.00
0.02
29_c <= 0.00
0.02
51_a <= 0.00
0.02
54_a > 0.00
0.02
53_g <= 0.00
0.02
56_g <= 0.00
0.02
20_g <= 0.00
0.02


0_g > 0.00
0.02
15_g <= 0.00
0.02
0_t <= 0.00
0.02
45_t <= 0.00
0.02
44_a <= 0.00
0.02
32_c <= 0.00
0.02
54_c <= 0.00
0.02
22_g <= 0.00
0.01
0.00 < 36_t <= 1.00
0.01
55_g <= 0.00
0.01
16_c <= 0.00
0.01
44_c <= 0.00
0.01
56_a <= 0.00
0.01
41_g <= 0.00
0.01
13_g <= 0.00
0.01
27_a <= 0.00
0.01
44_g <= 0.00
0.01
28_c <= 0.00
0.01
25_c <= 0.00
0.01
34_c <= 0.00
0.01
0.00 < 37_g <= 1.00
0.01
11_a > 0.00
0.01
7_g <= 0.00
0.01
37_a <= 0.00
0.01
49_g <= 0.00
0.01
41_a > 0.00
0.01
31_t > 0.00
0.01
12_g <= 0.00
0.01
26_g > 0.50
0.01
5_c > 0.00


0.01
42_a <= 0.00
0.01
2_a <= 0.00
0.01
53_a > 0.00
0.01
0.00 < 6_t <= 1.00
0.01
42_t <= 0.00
0.01
49_t <= 0.00
0.01
23_g > 0.00
0.01
52_g <= 0.00
0.01
46_g <= 0.00
0.01
3_t > 0.00
0.01
11_c <= 0.00
0.01
21_g <= 0.00
0.01
35_a <= 0.00
0.01
24_a <= 0.00
0.01
33_c > 0.00
0.01
3_g <= 0.00
0.01
19_c <= 0.00
0.01
16_t <= 0.00
0.01
29_t <= 0.00
0.00
23_a <= 0.00
0.00
0.00 < 24_t <= 1.00
0.00
31_g <= 0.00
0.00
0.00 < 28_t <= 1.00
0.00
12_c <= 0.00
0.00
5_t <= 0.00
0.00
21_t > 0.50
0.00
12_t <= 0.00
0.00
0.00 < 10_a <= 1.00
0.00
47_a > 0.00
0.00



Feature	Value
4_a	0.00
15_t	1.00
14_t	0.00
14_a	0.00
45_c	0.00
5_a	0.00
42_g	0.00
19_a	0.00
26_t	0.00
42_c	1.00

Conclusions

This data set was amenable to high predictive accuracy (92.6% for test accuracy with multiple models) and AUC-ROC (100% for logistic regression on test data) for ascertainment of promoter status using the included DNA sequences. Model interpretability was easy to obtain for individual nucleotide positions and identities, and though analytical methods differed, two regions of sequence appeared to have positions that were important (positions 14-19 and around 37-39). Bacterial -35 and -10 promoter elements are nucleotide hexamers, but many promoters have sequences that depart from consensus sequence; it is not always ideal for RNA polymerase to remain tightly bound to a portion of DNA. For each position ranked as important here, the presence of one nucleotide and/or absence of another nucleotide was regarded as impactful for promoter status by each analysis type. Between SHAP and LIME interpretability analyses on the logistic regression model, nucleotide identity did not always show the same predictions, but the relative importances of positions were important. Also, some positions, such as position 5, ranked highly for some reason; either this is due to a low number of samples or is indicative of another DNA element that is relevant to expression.

References

[1]Harley CB, Reynolds RP. Analysis of *E. coli* promoter sequences. *Nucleic Acids Res.* 1987;15:2343–2361.

[2]Noordewier M, Shavlik J, Harley C, Reynolds R. Molecular Biology (Promoter Gene Sequences) Data Set. https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+%28Promoter+Gene+Sequences%29 (https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+%28Promoter+Gene+Sequences%29 Dua D, Karra Taniskidou E. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml] (http://archive.ics.uci.edu/ml%5D). Irvine, CA: University of California, School of Information and Computer Science. 2017.

[3]Towell G, Shavlik J, Noordewier M. Refinement of Approximate Domain Theories by Knowledge-Based Artificial Neural Networks. *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*. 1990;861-866.

[4]Lundberg S. Shap. https://github.com/slundberg/shap (https://github.com/slundberg/shap).

[5]Ribeiro MT. Lime: Explaining the predictions of any machine learning classifier. https://github.com/marcotcr/lime (https://github.com/marcotcr/lime).