In [1]:

```
from __future__ import print_function
import sklearn
import sklearn.datasets
import sklearn.ensemble
import pandas as pd
import numpy as np
import lime
import lime.lime_tabular
import h2o
from h2o.estimators.random_forest import H2ORandomForestEstimator
from h2o.estimators.gbm import H2OGradientBoostingEstimator

np.random.seed(1)
```

In [2]:

```
# Start an H2O virtual cluster that uses 6 gigs of RAM and 6 cores
h2o.init(max_mem_size = "500M", nthreads = 6)

# Clean up the cluster just in case
h2o.remove_all()
```

```
Checking whether there is an H2O instance running at http://localhost:5432
1. connected.
```

H2O cluster uptime:	2 mins 30 secs
H2O cluster version:	3.10.4.8
H2O cluster version age:	16 days
H2O cluster name:	H2O_from_python_marcotcr_ph8pt1
H2O cluster total nodes:	1
H2O cluster free memory:	5.314 Gb
H2O cluster total cores:	4
H2O cluster allowed cores:	4
H2O cluster status:	locked, healthy
H2O connection url:	http://localhost:54321
H2O connection proxy:	None
H2O internal security:	False
Python version:	2.7.13 final

Wrapper class

We need a wrapper class that makes an H2O distributed random forest behave like a scikit learn random forest. We instantiate the class with an h2o distributed random forest object and column names. The predict_proba method takes a numpy array as input and returns an array of predicted probabilities for each class.

In [3]:

```
class h2o_predict_proba_wrapper:
    # drf is the h2o distributed random forest object, the column_names is the
    # labels of the X values
    def __init__(self,model,column_names):

            self.model = model
            self.column_names = column_names

    def predict_proba(self,this_array):
        # If we have just 1 row of data we need to reshape it
        shape_tuple = np.shape(this_array)
        if len(shape_tuple) == 1:
            this_array = this_array.reshape(1, -1)

        # We convert the numpy array that Lime sends to a pandas dataframe and
        # convert the pandas dataframe to an h2o frame
        self.pandas_df = pd.DataFrame(data = this_array,columns = self.column_names)
        self.h2o_df = h2o.H2OFrame(self.pandas_df)

        # Predict with the h2o drf
        self.predictions = self.model.predict(self.h2o_df).as_data_frame()
        # the first column is the class labels, the rest are probabilities for
        # each class
        self.predictions = self.predictions.iloc[:,1:].as_matrix()
        return self.predictions
```

Continuous features

Loading data, training a model

For this part, we'll use the Iris dataset, and we'll replace the scikit learn random forest with an h2o distributed random forest.

In [4]:

```
iris = sklearn.datasets.load_iris()

# Get the text names for the features and the class labels
feature_names = iris.feature_names
class_labels = 'species'
```

In [5]:

```
# Generate a train test split and convert to pandas and h2o frames

train, test, labels_train, labels_test = sklearn.model_selection.train_test_split(iris.
data, iris.target, train_size=0.80)

train_h2o_df = h2o.H2OFrame(train)
train_h2o_df.set_names(iris.feature_names)
train_h2o_df['species'] = h2o.H2OFrame(iris.target_names[labels_train])
train_h2o_df['species'] = train_h2o_df['species'].asfactor()

test_h2o_df = h2o.H2OFrame(test)
test_h2o_df.set_names(iris.feature_names)
test_h2o_df['species'] = h2o.H2OFrame(iris.target_names[labels_test])
test_h2o_df['species'] = test_h2o_df['species'].asfactor()
```

```
Parse progress: |████████████████████████████████████████████████████████
| 100%
Parse progress: |████████████████████████████████████████████████████████
| 100%
Parse progress: |████████████████████████████████████████████████████████
| 100%
Parse progress: |████████████████████████████████████████████████████████
| 100%
```

In [6]:

```
# rf = sklearn.ensemble.RandomForestClassifier(n_estimators=500)
# rf.fit(train, labels_train)

iris_drf = H2ORandomForestEstimator(
        model_id="iris_drf",
        ntrees=500,
        stopping_rounds=2,
        score_each_iteration=True,
        seed=1000000,
        balance_classes=False,
        histogram_type="AUTO")

iris_drf.train(x=feature_names,
         y='species',
         training_frame=train_h2o_df)
```

```
drf Model Build progress: |███████████████████████████████████████████████
| 100%
```

In [7]:

```
# sklearn.metrics.accuracy_score(labels_test, rf.predict(test))

iris_drf.model_performance(test_h2o_df)
```

```
ModelMetricsMultinomial: drf
** Reported on test data. **

MSE: 0.0270187026781
RMSE: 0.164373667837
LogLoss: 0.0874284925626
Mean Per-Class Error: 0.025641025641
Confusion Matrix: vertical: actual; across: predicted
```

setosa	versicolor	virginica	Error	Rate
11.0	0.0	0.0	0.0	0 / 11
0.0	12.0	1.0	0.0769231	1 / 13
0.0	0.0	6.0	0.0	0 / 6
11.0	12.0	7.0	0.0333333	1 / 30

Top-3 Hit Ratios:

k	hit_ratio
1	0.9666666
2	1.0
3	1.0

Out[7]:

Convert h2o to numpy array

The explainer requires numpy arrays as input and h2o requires the train and test data to be in h2o frames. In this case we could just use the train and test numpy arrays but for illustrative purposes here is how to convert an h2o frame to a pandas dataframe and a pandas dataframe to a numpy array.

In [8]:

```
train_pandas_df = train_h2o_df[feature_names].as_data_frame()
train_numpy_array = train_pandas_df.as_matrix()

test_pandas_df = test_h2o_df[feature_names].as_data_frame()
test_numpy_array = test_pandas_df.as_matrix()
```

Create the explainer

As opposed to lime_text.TextExplainer, tabular explainers need a training set. The reason for this is because we compute statistics on each feature (column). If the feature is numerical, we compute the mean and std, and discretize it into quartiles. If the feature is categorical, we compute the frequency of each value. For this tutorial, we'll only look at numerical features.

We use these computed statistics for two things:

1. To scale the data, so that we can meaningfully compute distances when the attributes are not on the same scale
2. To sample perturbed instances - which we do by sampling from a Normal(0,1), multiplying by the std and adding back the mean.

In [9]:

```
#explainer = lime.lime_tabular.LimeTabularExplainer(train, feature_names=iris.feature_n
ames, class_names=iris.target_names, discretize_continuous=True)

explainer = lime.lime_tabular.LimeTabularExplainer(train_numpy_array,
                                                   feature_names=feature_names,
                                                   class_names=iris.target_names,
                                                   discretize_continuous=True)
```

Create the predictor wrapper instance for the h2o drf

We have a trained h2o distributed random forest that we would like to explain. We need to create a wrapper instance that will make it behave as a scikit learn random forest for our Lime explainer.

In [10]:

```
h2o_drf_wrapper = h2o_predict_proba_wrapper(iris_drf,feature_names)
```

Explaining an instance

Since this is a multi-class classification problem, we set the top_labels parameter, so that we only explain the top class.

In [11]:

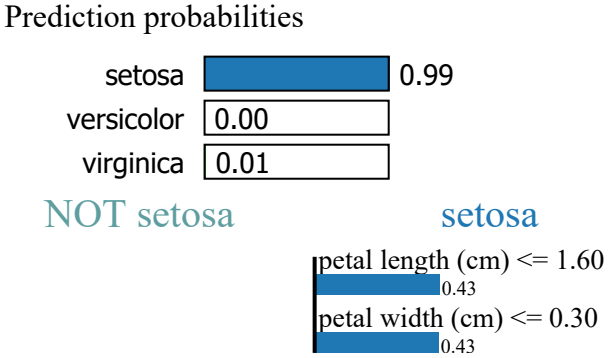
```
# i = np.random.randint(0, test_numpy_array.shape[0])

i = 27
# exp = explainer.explain_instance(test[i], rf.predict_proba, num_features=2, top_label
s=1)

exp = explainer.explain_instance(test_numpy_array[i], h2o_drf_wrapper.predict_proba, nu
m_features=2, top_labels=1)
```

```
Parse progress: |████████████████████████████████████████████████████████
| 100%
drf prediction progress: |████████████████████████████████████████████████
| 100%
```

We now explain a single instance:

In [12]:

```
exp.show_in_notebook(show_table=True, show_all=False)
```



Feature	Value
petal length (cm)	1.50
petal width (cm)	0.30

Now, there is a lot going on here. First, note that the row we are explained is displayed on the right side, in table format. Since we had the show_all parameter set to false, only the features used in the explanation are displayed.

The *value* column displays the original value for each feature.

Note that LIME has discretized the features in the explanation. This is because we let discretize_continuous=True in the constructor (this is the default). Discretized features make for more intuitive explanations.

Checking the local linear approximation

In [13]:

```
feature_index = lambda x: iris.feature_names.index(x)

print(test[i])
print(test_numpy_array[i])

# hide the parse and predict progress bars
h2o.no_progress()
```

```
[ 5.1  3.8  1.5  0.3]
[ 5.1  3.8  1.5  0.3]
```

In [14]:

```
print('Increasing petal width')
temp = test_numpy_array[i].copy()
print('P(setosa) before:', h2o_drf_wrapper.predict_proba(temp)[0,0])
temp[feature_index('petal width (cm)')] = 1.5
print('P(setosa) after:', h2o_drf_wrapper.predict_proba(temp)[0,0])
print ()
print('Increasing petal length')
temp = test_numpy_array[i].copy()
print('P(setosa) before:', h2o_drf_wrapper.predict_proba(temp)[0,0])
temp[feature_index('petal length (cm)')] = 3.5
print('P(setosa) after:', h2o_drf_wrapper.predict_proba(temp)[0,0])
print()
print('Increasing both')
temp = test_numpy_array[i].copy()
print('P(setosa) before:', h2o_drf_wrapper.predict_proba(temp)[0,0])
temp[feature_index('petal width (cm)')] = 1.5
temp[feature_index('petal length (cm)')] = 3.5
print('P(setosa) after:', h2o_drf_wrapper.predict_proba(temp)[0,0])
```

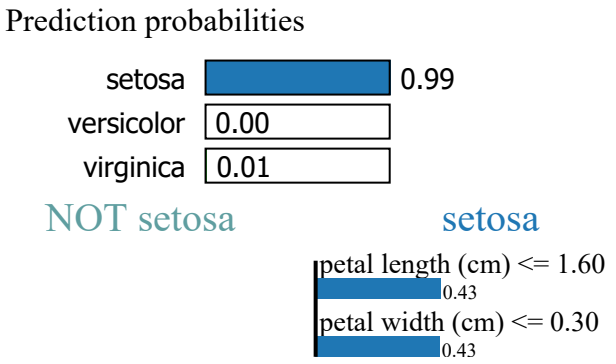
```
Increasing petal width
P(setosa) before: 0.993871818745
P(setosa) after: 0.523090430918

Increasing petal length
P(setosa) before: 0.993871818745
P(setosa) after: 0.523090430918

Increasing both
P(setosa) before: 0.993871818745
P(setosa) after: 0.0523090430918
```

Note that both features had the impact we thought they would. The scale at which they need to be perturbed of course depends on the scale of the feature in the training set.

We now show all features, just for completeness:

In [15]:

```
exp.show_in_notebook(show_table=True, show_all=True)
```



Feature	Value
sepal length (cm)	5.10
sepal width (cm)	3.80
petal length (cm)	1.50
petal width (cm)	0.30

Categorical features

For this part, we will use the Mushroom dataset, which will be downloaded here (http://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.data). The task is to predict if a mushroom is edible or poisonous, based on categorical features.

Loading data

In [16]:

```
# data = np.genfromtxt('http://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.data', delimiter=',', dtype='<U20')
data = np.genfromtxt('data/mushroom_data.csv', delimiter=',', dtype='<U20')
labels = data[:,0]
le= sklearn.preprocessing.LabelEncoder()
le.fit(labels)
labels = le.transform(labels)
class_names = le.classes_
data = data[:,1:]
```

In [17]:

```
categorical_features = range(22)
```

In [18]:

```
feature_names = 'cap-shape,cap-surface,cap-color,bruises?,odor,gill-attachment,gill-spacing,gill-size,gill-color,stalk-shape,stalk-root,stalk-surface-above-ring, stalk-surface-below-ring, stalk-color-above-ring,stalk-color-below-ring,veil-type,veil-color,ring-number,ring-type,spore-print-color,population,habitat'.split(',')
```

We expand the characters into words, using the data available in the UCI repository

In [19]:

```
categorical_names = '''bell=b,conical=c,convex=x,flat=f,knobbed=k,sunken=s
fibrous=f,grooves=g,scaly=y,smooth=s
brown=n,buff=b,cinnamon=c,gray=g,green=r,pink=p,purple=u,red=e,white=w,yellow=y
bruises=t,no=f
almond=a,anise=l,creosote=c,fishy=y,foul=f,musty=m,none=n,pungent=p,spicy=s
attached=a,descending=d,free=f,notched=n
close=c,crowded=w,distant=d
broad=b,narrow=n
black=k,brown=n,buff=b,chocolate=h,gray=g,green=r,orange=o,pink=p,purple=u,red=e,white=
w,yellow=y
enlarging=e,tapering=t
bulbous=b,club=c,cup=u,equal=e,rhizomorphs=z,rooted=r,missing=?
fibrous=f,scaly=y,silky=k,smooth=s
fibrous=f,scaly=y,silky=k,smooth=s
brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
partial=p,universal=u
brown=n,orange=o,white=w,yellow=y
none=n,one=o,two=t
cobwebby=c,evanescent=e,flaring=f,large=l,none=n,pendant=p,sheathing=s,zone=z
black=k,brown=n,buff=b,chocolate=h,green=r,orange=o,purple=u,white=w,yellow=y
abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y
grasses=g,leaves=l,meadows=m,paths=p,urban=u,waste=w,woods=d'''.split('\n')
for j, names in enumerate(categorical_names):
    values = names.split(',')
    values = dict([(x.split('=')[1], x.split('=')[0]) for x in values])
    data[:,j] = np.array(list(map(lambda x: values[x], data[:,j])))
```

Our explainer (and most classifiers) takes in numerical data, even if the features are categorical. We thus transform all of the string attributes into integers, using sklearn's LabelEncoder. We use a dict to save the correspondence between the integer values and the original strings, so that we can present this later in the explanations.

The h2o drf classifier can handle the original text, but we will convert to integers to Lime and then have h20 treat the categories as strings.

In [20]:

```
categorical_names = {}
for feature in categorical_features:
    le = sklearn.preprocessing.LabelEncoder()
    le.fit(data[:, feature])
    data[:, feature] = le.transform(data[:, feature])
    categorical_names[feature] = le.classes_
```

In [21]:

```
data[:,0]
```

Out[21]:

```
array([u'2', u'2', u'0', ..., u'3', u'4', u'2'],
      dtype='<U20')
```

In [22]:

```
categorical_names[0]
```

Out[22]:

```
array([u'bell', u'conical', u'convex', u'flat', u'knobbed', u'sunken'],
      dtype='<U20')
```

We now split the data into training and testing

In [23]:

```
data = data.astype(float)
```

In [24]:

```
# Generate a train test split and convert to pandas and h2o frames

train, test, labels_train, labels_test = sklearn.model_selection.train_test_split(data,
labels, train_size=0.80)

class_names=['edible', 'poisonous']

train_h2o_df = h2o.H2OFrame(train)
train_h2o_df.set_names(feature_names)
train_h2o_df['class'] = h2o.H2OFrame(labels_train)
train_h2o_df['class'] = train_h2o_df['class'].asfactor()

test_h2o_df = h2o.H2OFrame(test)
test_h2o_df.set_names(feature_names)
test_h2o_df['class'] = h2o.H2OFrame(labels_test)
test_h2o_df['class'] = test_h2o_df['class'].asfactor()
```

~~Finally, we use a One-hot encoder, so that the classifier does not take our categorical features as continuous features. We will use this encoder only for the classifier, not for the explainer - and the reason is that the explainer must make sure that a categorical feature only has one value.~~

A great feature of h2o is that it can handle categorical data directly. There is no need to one-hot encode. We will use the integer data for the explainer and have h2o treat the integers as text.

We do have to tell h2o that the integer values in the data should be treated as categories and not numeric values. ".asfactor()" does this.

In [25]:

```
for feature in categorical_features:
    train_h2o_df[feature] = train_h2o_df[feature].asfactor()
    test_h2o_df[feature] = test_h2o_df[feature].asfactor()
```

In [26]:

```
# encoder = sklearn.preprocessing.OneHotEncoder(categorical_features=categorical_featur
es)
```

In [27]:

```
# encoder.fit(data)
# encoded_train = encoder.transform(train)
```

In [28]:

```
# rf = sklearn.ensemble.RandomForestClassifier(n_estimators=500)
# rf.fit(encoded_train, labels_train)

# Train an h2o drf
mushroom_drf = H2ORandomForestEstimator(
                model_id="mushroom_drf",
                ntrees=500,
                stopping_rounds=2,
                score_each_iteration=True,
                seed=1000000,
                balance_classes=False,
                histogram_type="AUTO")

mushroom_drf.train(x=feature_names,
             y='class',
             training_frame=train_h2o_df)
```

~~Note that our predict function first transforms the data into the one-hot representation~~

Create a predictor object

In [29]:

```
# predict_fn = lambda x: rf.predict_proba(encoder.transform(x))

h2o_drf_wrapper = h2o_predict_proba_wrapper(mushroom_drf,feature_names)
```

This classifier has perfect accuracy on the test set!

In [30]:

```
# sklearn.metrics.accuracy_score(labels_test, rf.predict(encoder.transform(test)))

mushroom_drf.model_performance(test_h2o_df)
```

```
ModelMetricsBinomial: drf
** Reported on test data. **

MSE: 6.62117237387e-05
RMSE: 0.00813705866629
LogLoss: 0.000860149849105
Mean Per-Class Error: 0.0
AUC: 1.0
Gini: 1.0
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.858024689886:
```

	0	1	Error	Rate
0	849.0	0.0	0.0	(0.0/849.0)
1	0.0	776.0	0.0	(0.0/776.0)
Total	849.0	776.0	0.0	(0.0/1625.0)

```
Maximum Metrics: Maximum metrics at their respective thresholds
```

metric	threshold	value	idx
max f1	0.8580247	1.0	3.0
max f2	0.8580247	1.0	3.0
max f0point5	0.8580247	1.0	3.0
max accuracy	0.8580247	1.0	3.0
max precision	1.0	1.0	0.0
max recall	0.8580247	1.0	3.0
max specificity	1.0	1.0	0.0
max absolute_mcc	0.8580247	1.0	3.0
max min_per_class_accuracy	0.8580247	1.0	3.0
max mean_per_class_accuracy	0.8580247	1.0	3.0

```
Gains/Lift Table: Avg response rate: 47.75 %
```

	group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	respo
	1	0.4695385	1.0	2.0940722	2.0940722	1.0
	2	1.0	0.0	0.0315811	1.0	0.015

```
Out[30]:
```

Explaining predictions

We now create our explainer. The categorical_features parameter lets it know which features are categorical (in this case, all of them). The categorical names parameter gives a string representation of each categorical feature's numerical value, as we saw before.

In [31]:

```
np.random.seed(1)
```

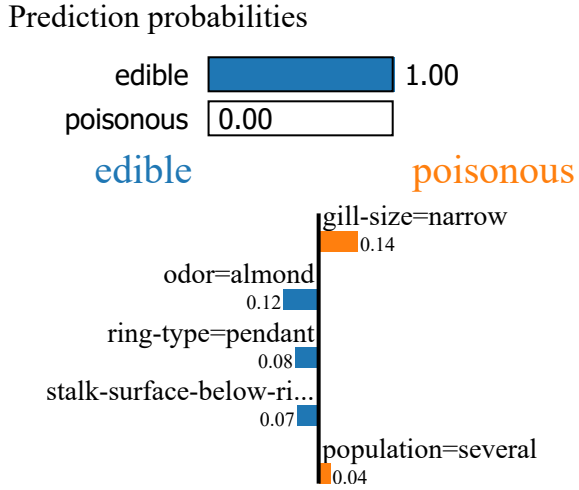
In [32]:

```
explainer = lime.lime_tabular.LimeTabularExplainer(train ,class_names=class_names, feat
ure_names = feature_names,
                                                   categorical_features=categorical_fea
tures,
                                                   categorical_names=categorical_names,
kernel_width=3, verbose=True)
```

In [33]:

```
i = 137
exp = explainer.explain_instance(test[i], h2o_drf_wrapper.predict_proba, num_features=5
)
exp.show_in_notebook()
```

```
Intercept 0.488622041715
Prediction_local [ 0.38943212]
Right: 0.0
```



Feature	Value
gill-size=narrow	True
odor=almond	True
ring-type=pendant	True
stalk-surface-below-ring=smooth	True
population=several	True

Now note that the explanations are based not only on features, but on feature-value pairs. For example, we are saying that odor=foul is indicative of a poisonous mushroom. In the context of a categorical feature, odor could take many other values (see below). Since we perturb each categorical feature drawing samples according to the original training distribution, the way to interpret this is: if odor was not foul, on average, this prediction would be 0.24 less 'poisonous'. Let's check if this is the case

In [34]:

```
odor_idx = feature_names.index('odor')
explainer.categorical_names[odor_idx]
```

Out[34]:

```
array([u'almond', u'anise', u'creosote', u'fishy', u'foul', u'musty',
       u'none', u'pungent', u'spicy'],
      dtype='<U20')
```

In [35]:

```
explainer.feature_frequencies[odor_idx]
```

Out[35]:

```
array([ 0.04985382,  0.0489306 ,  0.02384982,  0.07031851,  0.26450223,
        0.00384675,  0.43375904,  0.03277427,  0.07216495])
```

In [36]:

```
foul_idx = 4
non_foul = np.delete(explainer.categorical_names[odor_idx], foul_idx)
non_foul_normalized_frequencies = explainer.feature_frequencies[odor_idx].copy()
non_foul_normalized_frequencies[foul_idx] = 0
non_foul_normalized_frequencies /= non_foul_normalized_frequencies.sum()
```

In [37]:

```
print('Making odor not equal foul')
temp = test[i].copy()
print('P(poisonous) before:', h2o_drf_wrapper.predict_proba(temp)[0,1])
print
average_poisonous = 0
for idx, (name, frequency) in enumerate(zip(explainer.categorical_names[odor_idx], non_
foul_normalized_frequencies)):
    if name == 'foul':
        continue
    temp[odor_idx] = idx
    p_poisonous = h2o_drf_wrapper.predict_proba(temp)[0,1]
    average_poisonous += p_poisonous * frequency
    print('P(poisonous | odor=%s): %.2f' % (name, p_poisonous))
print ()
print ('P(poisonous | odor != foul) = %.2f' % average_poisonous)
```

```
Making odor not equal foul
P(poisonous) before: 0
P(poisonous | odor=almond): 0.00
P(poisonous | odor=anise): 0.00
P(poisonous | odor=creosote): 0.44
P(poisonous | odor=fishy): 0.33
P(poisonous | odor=musty): 0.33
P(poisonous | odor=none): 0.00
P(poisonous | odor=pungent): 0.44
P(poisonous | odor=spicy): 0.33

P(poisonous | odor != foul) = 0.10
```

We see that in this particular case, the linear model is pretty close: it predicted that on average odor increases the probability of poisonous by 0.26, when in fact it is by 0.23. Notice though that we only changed one feature (odor), when the linear model takes into account perturbations of all the features at once.

Numerical and Categorical features in the same dataset

We now turn to a dataset that has both numerical and categorical features. Here, the task is to predict whether a person makes over 50K dollars per year. Downloads the data here (http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data).

In [38]:

```
feature_names = ["Age", "Workclass", "fnlwgt", "Education", "Education-Num", "Marital S
tatus","Occupation", "Relationship", "Race", "Sex", "Capital Gain", "Capital Loss","Hou
rs per week", "Country"]
```

In [39]:

```
# data = np.genfromtxt('http://archive.ics.uci.edu/ml/machine-learning-databases/adult/
adult.data', delimiter=', ', dtype=str)
data = np.genfromtxt('data/adult.csv', delimiter=', ', dtype=str)
```

In [40]:

```
labels = data[:,14]
le= sklearn.preprocessing.LabelEncoder()
le.fit(labels)
labels = le.transform(labels)
class_names = le.classes_
data = data[:,:-1]
```

In [41]:

```
categorical_features = [1,3,5, 6,7,8,9,13]
```

In [42]:

```
categorical_names = {}
for feature in categorical_features:
    le = sklearn.preprocessing.LabelEncoder()
    le.fit(data[:, feature])
    data[:, feature] = le.transform(data[:, feature])
    categorical_names[feature] = le.classes_
```

In [43]:

```
data = data.astype(float)
```

In [44]:

```
# We don't need to one-hot encode from h2o
# encoder = sklearn.preprocessing.OneHotEncoder(categorical_features=categorical_featur
es)
```

In [45]:

```
# Generate a train test split and convert to pandas and h2o frames

np.random.seed(1)
train, test, labels_train, labels_test = sklearn.model_selection.train_test_split(data,
labels, train_size=0.80)


train_h2o_df = h2o.H2OFrame(train)
train_h2o_df.set_names(feature_names)
train_h2o_df['class'] = h2o.H2OFrame(labels_train)
train_h2o_df['class'] = train_h2o_df['class'].asfactor()

test_h2o_df = h2o.H2OFrame(test)
test_h2o_df.set_names(feature_names)
test_h2o_df['class'] = h2o.H2OFrame(labels_test)
test_h2o_df['class'] = test_h2o_df['class'].asfactor()

print(type(train_h2o_df))
```

```
<class 'h2o.frame.H2OFrame'>
```

In [46]:

```
# We don't need to one-hot encode from h2o
# encoder.fit(data)
# encoded_train = encoder.transform(train)
```

We need to tell h2o which features are categorical.

In [47]:

```
for feature in categorical_features:
    train_h2o_df[feature] = train_h2o_df[feature].asfactor()
    test_h2o_df[feature] = test_h2o_df[feature].asfactor()
```

~~This time, we use gradient boosted trees as the model, using the xgboost (https://github.com/dmlc/xgboost) package.~~

We will use the H2OGradientBoostingEstimator which is h2o's version of gradient boosted trees.

In [48]:

```
adult_gbm = H2OGradientBoostingEstimator(
            ntrees=300,
            learn_rate=0.1,
            max_depth=5,
            stopping_tolerance=0.01,
            stopping_rounds=2,
            score_each_iteration=True,
            model_id="gbm_v1",
            seed=2000000
)
adult_gbm.train(feature_names, 'class', training_frame = train_h2o_df)
```

In [49]:

```
# sklearn.metrics.accuracy_score(labels_test, rf.predict(encoder.transform(test)))

adult_gbm.model_performance(test_h2o_df)
```

```
ModelMetricsBinomial: gbm
** Reported on test data. **

MSE: 0.0926845991628
RMSE: 0.304441454409
LogLoss: 0.29897266577
Mean Per-Class Error: 0.167136137545
AUC: 0.917242377298
Gini: 0.834484754596
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.402214919768:
```

	0	1	Error	Rate
0	4581.0	445.0	0.0885	(445.0/5026.0)
1	440.0	1047.0	0.2959	(440.0/1487.0)
Total	5021.0	1492.0	0.1359	(885.0/6513.0)

```
Maximum Metrics: Maximum metrics at their respective thresholds
```

metric	threshold	value	idx
max f1	0.4022149	0.7029204	187.0
max f2	0.1984941	0.7859748	276.0
max f0point5	0.6303836	0.7411482	104.0
max accuracy	0.4960229	0.8705666	152.0
max precision	0.9599827	1.0	0.0
max recall	0.0212204	1.0	396.0
max specificity	0.9599827	1.0	0.0
max absolute_mcc	0.4618071	0.6170370	164.0
max min_per_class_accuracy	0.2744757	0.8258238	240.0
max mean_per_class_accuracy	0.2276158	0.8328639	260.0

```
Gains/Lift Table: Avg response rate: 22.83 %
```

	group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	respo
	1	0.0101336	0.9424305	4.2472336	4.2472336	0.969
	2	0.0201136	0.9349005	4.3799597	4.3130900	1.0
	3	0.0300937	0.9287624	4.3799597	4.3352662	1.0
	4	0.0402272	0.9163701	4.3135966	4.3298074	0.984
	5	0.0500537	0.8705261	4.3115228	4.3262178	0.984
	6	0.1002610	0.7176037	3.5093255	3.9171461	0.801
	7	0.1500077	0.5813814	2.8253443	3.5550747	0.645
	8	0.2005220	0.4617318	2.4096435	3.2665243	0.550
	9	0.3000154	0.3027815	1.4464682	2.6629437	0.330
	10	0.3999693	0.1912673	1.0428475	2.2580752	0.238
	11	0.5002303	0.1057253	0.5500102	1.9157282	0.125
	12	0.6003378	0.0599257	0.2485560	1.6377240	0.056
	13	0.6999846	0.0382068	0.1282269	1.4228384	0.029
	14	0.7999386	0.0310427	0.0201841	1.2475739	0.004
	15	0.9012744	0.0241761	0.0132726	1.1087939	0.003
	16	1.0	0.0150221	0.0068118	1.0	0.001

Out[49]:

In [50]:

```
# predict_fn = lambda x: rf.predict_proba(encoder.transform(x)).astype(float)

h2o_drf_wrapper = h2o_predict_proba_wrapper(adult_gbm,feature_names)
```

Explaining predictions

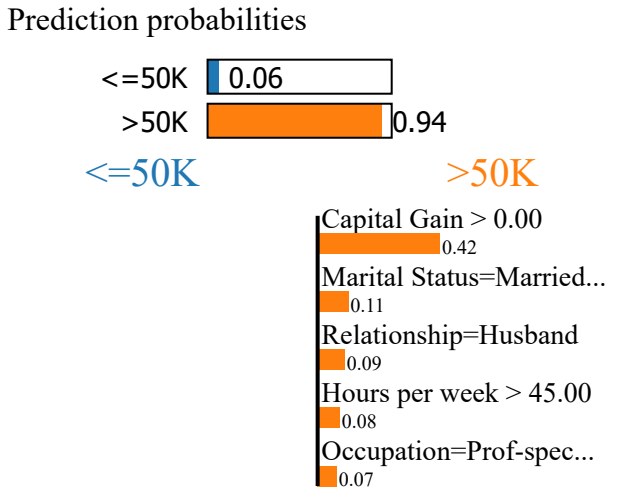
In [51]:

```
explainer = lime.lime_tabular.LimeTabularExplainer(train ,feature_names = feature_names
,class_names=class_names,
                                                   categorical_features=categorical_fea
tures,
                                                   categorical_names=categorical_names,
kernel_width=3)
```

We now show a few explanations. These are just a mix of the continuous and categorical examples we showed before. For categorical features, the feature contribution is always the same as the linear model weight.

In [52]:

```
np.random.seed(1)
i = 1653
exp = explainer.explain_instance(test[i], h2o_drf_wrapper.predict_proba, num_features=5
)
exp.show_in_notebook(show_all=False)
```

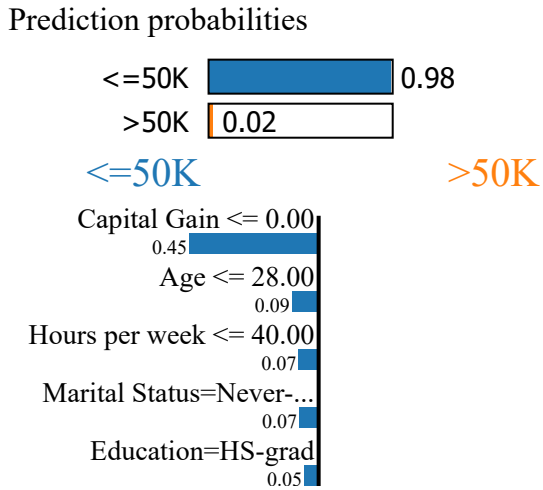


Feature	Value
Capital Gain	15024.00
Marital Status=Married-civ-spouse	True
Relationship=Husband	True
Hours per week	60.00
Occupation=Prof-specialty	True

Note that capital gain has very high weight. This makes sense. Now let's see an example where the person has a capital gain below the mean:

In [53]:

```
i = 10
exp = explainer.explain_instance(test[i], h2o_drf_wrapper.predict_proba, num_features=5
)
exp.show_in_notebook(show_all=False)
```



Feature	Value
Capital Gain	0.00
Age	19.00
Hours per week	30.00
Marital Status=Never-married	True
Education=HS-grad	True

In [54]:

```
h2o.shutdown(prompt=False)
```

```
[WARNING] in <ipython-input-54-06c455af4589> line 1:
    >>> h2o.shutdown(prompt=False)
        ^^^^ Deprecated, use ``h2o.cluster().shutdown()``.
H2O session _sid_a932 closed.
```