Random Forest

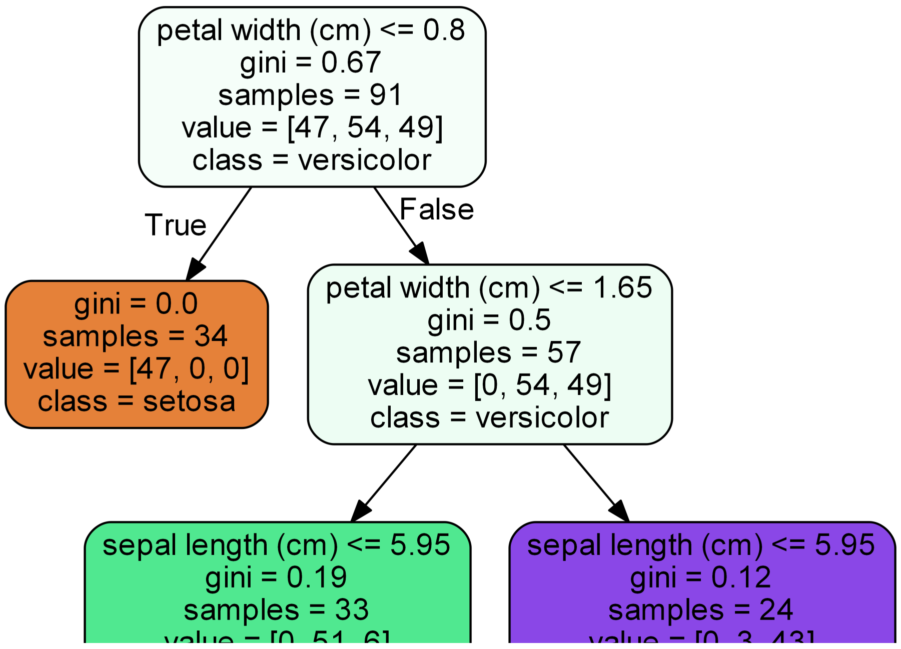
Random Forest is an ensemble of Decision Trees. With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

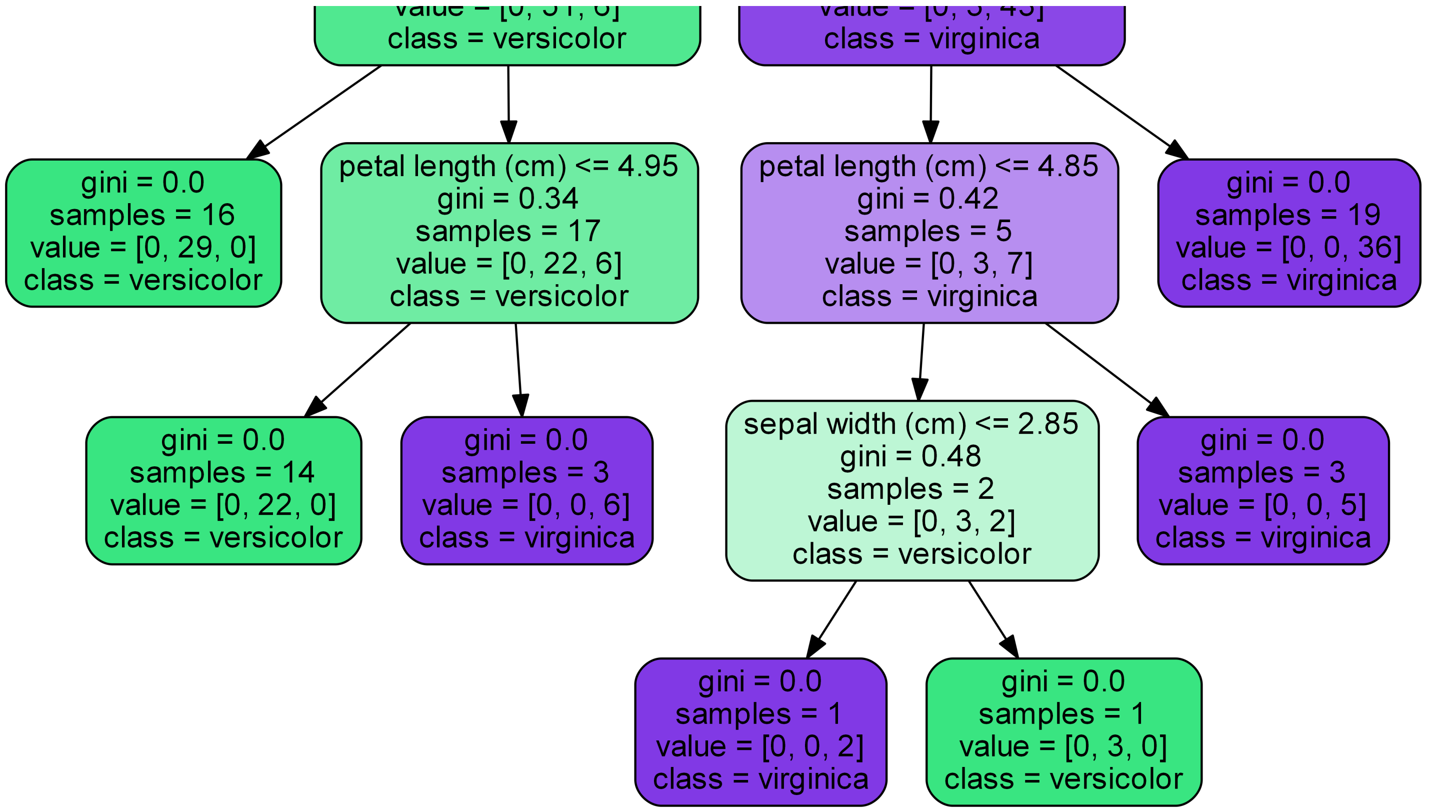
The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features. This results in a greater tree diversity, which (once again) trades a higher bias for a lower variance, generally yielding an overall better model. The following `BaggingClassifier` is roughly equivalent to the previous `RandomForestClassifier`. Run the cell below to visualize a single estimator from a random forest model, using the Iris dataset to classify the data into the appropriate species.

In [57]:

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

# Model (can also use single decision tree)
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=10)

# Train
model.fit(iris.data, iris.target)
# Extract single tree
estimator = model.estimators_[5]

from sklearn.tree import export_graphviz
# Export as dot file
export_graphviz(estimator, out_file='tree.dot',
                feature_names = iris.feature_names,
                class_names = iris.target_names,
                rounded = True, proportion = False,
                precision = 2, filled = True)

# Convert to png using system command (requires Graphviz)
from subprocess import call
call(['dot', '-Tpng', 'tree.dot', '-o', 'tree.png', '-Gdpi=600'])

# Display in jupyter notebook
from IPython.display import Image
Image(filename = 'tree.png')
```

Out[57]:





Notice how each split seperates the data into buckets of similar observations. This is a single tree and a relatively simple classification dataset, but the same method is used in a more complex dataset with greater depth to the trees.

Coronavirus

Coronavirus disease (COVID-19) is an infectious disease caused by a new virus. The disease causes respiratory illness (like the flu) with symptoms such as a cough, fever, and in more severe cases, difficulty breathing. You can protect yourself by washing your hands frequently, avoiding touching your face, and avoiding close contact (1 meter or 3 feet) with people who are unwell. An outbreak of COVID-19 started in December 2019 and at the time of the creation of this project was continuing to spread throughout the world. Many governments recommended only essential outings to public places and closed most business that do not serve food or sell essential items. An excellent spatial dashboard built by Johns Hopkins shows the daily confirmed cases by country.

This case study was designed to drive home the important role that data science plays in real-world situations like this pandemic. This case study uses the Random Forest Classifier and a dataset from the South Korean cases of COVID-19 provided on Kaggle to encourage research on this important topic. The goal of the case study is to build a Random Forest Classifier to predict the 'state' of the patient.

First, please load the needed packages and modules into Python. Next, load the data into a pandas dataframe for ease of use.

In [2]:

```
import os
import pandas as pd
from datetime import datetime,timedelta
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
import plotly.graph_objects as go
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import ExtraTreesRegressor
```

In [3]:

```
#url ='SouthKoreacoronavirusdataset/PatientInfo.csv'
df = pd.read_csv(r'C:\Users\admin\Desktop\SPRINGBOARDFILES\Unit 14\Unit 14.4\1593492590_R
andomForest_Covid_Case_Study_06302020\RandomForest Covid Case Study_06302020\PatientInfo.
csv')
df.head()
```

Out[3]:

	patient_id	global_num	sex	birth_year	age	country	province	city	disease	infection_case	infection_order	
0	1000000001	2.0	male	1964.0	50s	Korea	Seoul	Gangseo-gu	NaN	overseas inflow	1.0	
1	1000000002	5.0	male	1987.0	30s	Korea	Seoul	Jungnang-gu	NaN	overseas inflow	1.0	
2	1000000003	6.0	male	1964.0	50s	Korea	Seoul	Jongno-gu	NaN	contact with patient	2.0	2.
3	1000000004	7.0	male	1991.0	20s	Korea	Seoul	Mapo-gu	NaN	overseas inflow	1.0	
4	1000000005	9.0	female	1992.0	20s	Korea	Seoul	Seongbuk-gu	NaN	contact with patient	2.0	1.

In [4]:

```
df.shape
```

Out[4]:

```
(2218, 18)
```

In [5]:

```
df["disease"].unique()
```

Out[5]:

```
array([nan, True], dtype=object)
```

In [6]:

```
#Counts of null values
na_df=pd.DataFrame(df.isnull().sum().sort_values(ascending=False)).reset_index()
na_df.columns = ['VarName', 'NullCount']
na_df[(na_df['NullCount']>0)]
```

Out[6]:

	VarName	NullCount
0	disease	2199
1	deceased_date	2186
2	infection_order	2176
3	symptom_onset_date	2025
4	released_date	1995
5	contact_number	1807
6	infected_by	1749
7	infection_case	1055
8	global_num	904
9	birth_year	454
10	age	261
11	sex	145
12	confirmed_date	141

	VarName	NullCount
13	state	88
14	city	65

In [7]:

```
#counts of response variable values
df.state.value_counts()
```

Out[7]:

```
isolated    1791
released     307
deceased      32
Name: state, dtype: int64
```

Create a new column named 'n_age' which is the calculated age based on the birth year column.

In [8]:

```
df['n_age']=2020- df['birth_year']
df['n_age']
```

Out[8]:

```
0       56.0
1       33.0
2       56.0
3       29.0
4       28.0
        ...
2213    30.0
2214    22.0
2215    22.0
2216    48.0
2217    46.0
Name: n_age, Length: 2218, dtype: float64
```

Handle Missing Values

Print the number of missing values by column.

In [9]:

```
df.isnull().sum()
```

Out[9]:

```
patient_id               0
global_num             904
sex                    145
birth_year             454
age                    261
country                  0
province                 0
city                    65
disease               2199
infection_case        1055
infection_order       2176
infected_by           1749
contact_number        1807
symptom_onset_date    2025
confirmed_date         141
released_date         1995
deceased_date         2186
state                   88
n_age                  454
dtype: int64
```

In [10]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2218 entries, 0 to 2217
Data columns (total 19 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   patient_id          2218 non-null   int64  
 1   global_num          1314 non-null   float64
 2   sex                 2073 non-null   object 
 3   birth_year          1764 non-null   float64
 4   age                 1957 non-null   object 
 5   country             2218 non-null   object 
 6   province            2218 non-null   object 
 7   city                2153 non-null   object 
 8   disease             19 non-null     object 
 9   infection_case      1163 non-null   object 
 10  infection_order     42 non-null     float64
 11  infected_by         469 non-null    float64
 12  contact_number      411 non-null    float64
 13  symptom_onset_date  193 non-null    object 
 14  confirmed_date      2077 non-null   object 
 15  released_date       223 non-null    object 
 16  deceased_date       32 non-null     object 
 17  state               2130 non-null   object 
 18  n_age               1764 non-null   float64
dtypes: float64(6), int64(1), object(12)
memory usage: 225.3+ KB
```

Fill the 'disease' missing values with 0 and remap the True values to 1.

In [11]:

```
df["disease"] = df["disease"].fillna(0)
```

In [12]:

```
df["disease"] = df["disease"].replace(True,1)
```

In [13]:

```
df["disease"].unique()
```

Out[13]:

```
array([0, 1], dtype=int64)
```

Fill null values in the following columns with their mean: 'global_number','birth_year','infection_order','infected_by'and 'contact_number'

In [14]:

```
df['global_num'].fillna((df['global_num'].mean()), inplace=True)
```

In [15]:

```
df['birth_year'].fillna((df['birth_year'].mean()), inplace=True)
```

In [16]:

```
df['infection_order'].fillna((df['infection_order'].mean()), inplace=True)
```

In [17]:

```
df['infected_by'].fillna((df['infected_by'].mean()), inplace=True)
```

In [18]:

```
df['contact_number'].fillna((df['contact_number'].mean()), inplace=True)
```

Fill the rest of the missing values with any method.

In [19]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2218 entries, 0 to 2217
Data columns (total 19 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   patient_id          2218 non-null   int64  
 1   global_num          2218 non-null   float64
 2   sex                 2073 non-null   object 
 3   birth_year          2218 non-null   float64
 4   age                 1957 non-null   object 
 5   country             2218 non-null   object 
 6   province            2218 non-null   object 
 7   city                2153 non-null   object 
 8   disease             2218 non-null   int64  
 9   infection_case      1163 non-null   object 
 10  infection_order     2218 non-null   float64
 11  infected_by         2218 non-null   float64
 12  contact_number      2218 non-null   float64
 13  symptom_onset_date  193 non-null    object 
 14  confirmed_date      2077 non-null   object 
 15  released_date       223 non-null    object 
 16  deceased_date       32 non-null     object 
 17  state               2130 non-null   object 
 18  n_age               1764 non-null   float64
dtypes: float64(6), int64(2), object(11)
memory usage: 234.0+ KB
```

In [20]:

```
df['sex'].fillna((df['sex'].mode()[0]), inplace=True)
```

In [21]:

```
df['age'].fillna((df['age'].mode()[0]), inplace=True)
```

In [22]:

```
df['city'].fillna((df['city'].mode()[0]), inplace=True)
```

In [23]:

```
df['infection_case'].fillna((df['infection_case'].mode()[0]), inplace=True)
```

In [24]:

```
df['symptom_onset_date'].fillna((df['symptom_onset_date'].mode()[0]), inplace=True)
```

In [25]:

```
df['confirmed_date'].fillna((df['confirmed_date'].mode()[0]), inplace=True)
```

In [26]:

```
df['deceased_date'].fillna((df['deceased_date'].mode()[0]), inplace=True)
```

In [27]:

```
df['state'].fillna((df['state'].mode()[0]), inplace=True)
```

In [28]:

```
df['n_age'].fillna((df['n_age'].mode()[0]), inplace=True)
```

Check for any remaining null values.

In [29]:

```
df.isnull()
```

Out[29]:

	patient_id	global_num	sex	birth_year	age	country	province	city	disease	infection_case	infection_order	infec
0	False	False	False	False	False	False	False	False	False	False	False	
1	False	False	False	False	False	False	False	False	False	False	False	
2	False	False	False	False	False	False	False	False	False	False	False	
3	False	False	False	False	False	False	False	False	False	False	False	
4	False	False	False	False	False	False	False	False	False	False	False	
...	...	...	...	...	...	...	...	...	...	...	...	
2213	False	False	False	False	False	False	False	False	False	False	False	
2214	False	False	False	False	False	False	False	False	False	False	False	
2215	False	False	False	False	False	False	False	False	False	False	False	
2216	False	False	False	False	False	False	False	False	False	False	False	
2217	False	False	False	False	False	False	False	False	False	False	False	

2218 rows × 19 columns

In [30]:

```
df.head()
```

Out[30]:

	patient_id	global_num	sex	birth_year	age	country	province	city	disease	infection_case	infection_order	
0	1000000001	2.0	male	1964.0	50s	Korea	Seoul	Gangseo-gu	0	overseas inflow	1.0	2.
1	1000000002	5.0	male	1987.0	30s	Korea	Seoul	Jungnang-gu	0	overseas inflow	1.0	2.
2	1000000003	6.0	male	1964.0	50s	Korea	Seoul	Jongno-gu	0	contact with patient	2.0	2.
3	1000000004	7.0	male	1991.0	20s	Korea	Seoul	Mapo-gu	0	overseas inflow	1.0	2.
4	1000000005	9.0	female	1992.0	20s	Korea	Seoul	Seongbuk-gu	0	contact with patient	2.0	1.

Remove date columns from the data.

In [31]:

```
df = df.drop(['symptom_onset_date','confirmed_date','released_date','deceased_date'],axi
s =1)
```

Review the count of unique values by column.

In [32]:

```
print(df.nunique())
```

```
patient_id          2218
global_num          1304
```

```
sex                   2
birth_year           97
age                  11
country               4
province             17
city                134
disease               2
infection_case       16
infection_order       7
infected_by         207
contact_number       73
state                 3
n_age                96
dtype: int64
```

Review the percent of unique values by column.

In [33]:

```
print(df.nunique()/df.shape[0])
```

```
patient_id         1.000000
global_num         0.587917
sex                0.000902
birth_year         0.043733
age                0.004959
country            0.001803
province           0.007665
city               0.060415
disease            0.000902
infection_case     0.007214
infection_order    0.003156
infected_by        0.093327
contact_number     0.032913
state              0.001353
n_age              0.043282
dtype: float64
```

Review the range of values per column.

In [34]:

```
df.describe().T
```

Out[34]:

	count	mean	std	min	25%	50%	75%	max
patient_id	2218.0	4.014678e+09	2.192419e+09	1.000000e+09	1.700000e+09	6.001000e+09	6.004000e+09	7.000000e+09
global_num	2218.0	4.664817e+03	2.211785e+03	1.000000e+00	4.205250e+03	4.664817e+03	5.900250e+03	8.717000e+03
birth_year	2218.0	1.974989e+03	1.731123e+01	1.916000e+03	1.965000e+03	1.974989e+03	1.988000e+03	2.020000e+03
disease	2218.0	8.566276e-03	9.217769e-02	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	1.000000e+00
infection_order	2218.0	2.285714e+00	1.706622e-01	1.000000e+00	2.285714e+00	2.285714e+00	2.285714e+00	6.000000e+00
infected_by	2218.0	2.600789e+09	7.216328e+08	1.000000e+09	2.600789e+09	2.600789e+09	2.600789e+09	6.113000e+09
contact_number	2218.0	2.412895e+01	3.917141e+01	0.000000e+00	2.412895e+01	2.412895e+01	2.412895e+01	1.160000e+03
n_age	2218.0	4.623715e+01	1.747912e+01	0.000000e+00	3.200000e+01	5.100000e+01	5.500000e+01	1.040000e+02

Check for duplicated rows

In [35]:

```
duplicateRowsDF = df[df.duplicated()]
duplicateRowsDF
```

Out[35]:

	patient_id	global_num	sex	birth_year	age	country	province	city	disease	infection_case	infection_order	infected_by	c

Print the categorical columns and their associated levels.

In [36]:

```
dfo = df.select_dtypes(include=['object'], exclude=['datetime'])
dfo.shape
#get levels for all variables
vn = pd.DataFrame(dfo.nunique()).reset_index()
vn.columns = ['VarName', 'LevelsCount']
vn.sort_values(by='LevelsCount', ascending =False)
vn
```

Out[36]:

	VarName	LevelsCount
0	sex	2
1	age	11
2	country	4
3	province	17
4	city	134
5	infection_case	16
6	state	3

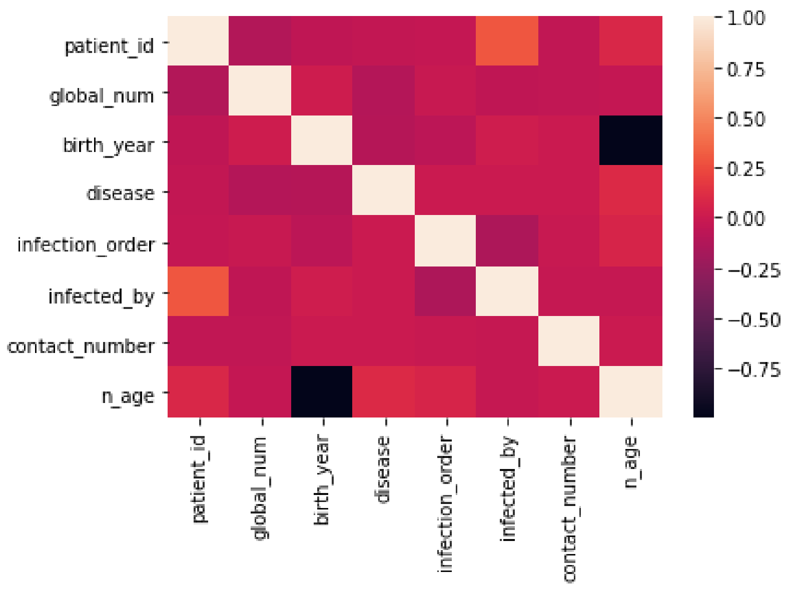
Plot the correlation heat map for the features.

In [37]:

```
sns.heatmap(df.corr())
```

Out[37]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x8ffe2d0>
```



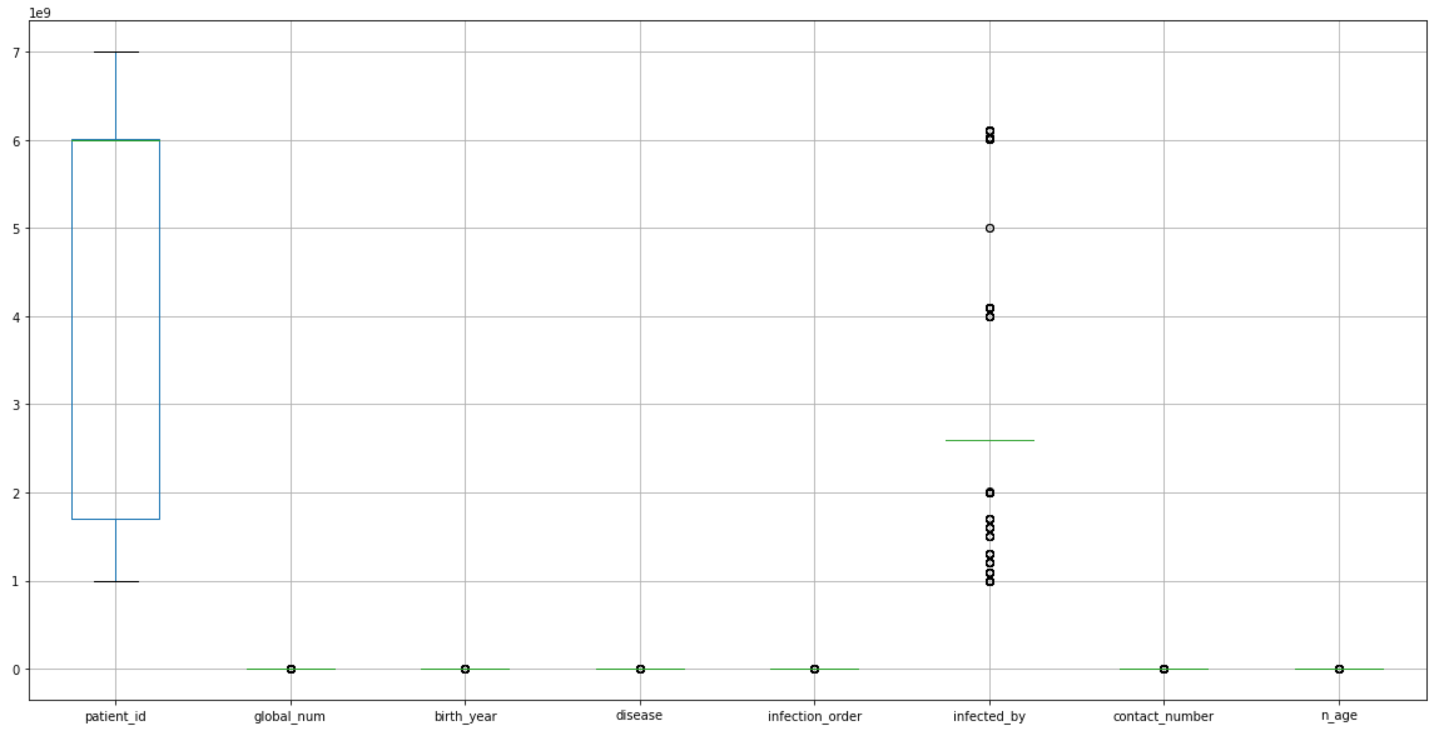
Plot the boxplots to check for outliers.

In [38]:

```
plt.figure(figsize=(20,10))
df.boxplot()
```

Out[38]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x944e170>
```



Create dummy features for object type features.

In [39]:

```
features=['sex','age','country','province', 'city','infection_case']
dummies=pd.get_dummies(df[features])
merged=pd.concat([df,dummies],axis=1)
final=merged.drop(['sex','age','country','province', 'city','infection_case'], axis=1)
df=final
df.head()
```

Out[39]:

	patient_id	global_num	birth_year	disease	infection_order	infected_by	contact_number	state	n_age	sex_female
0	1000000001	2.0	1964.0	0	1.0	2.600789e+09	75.0	released	56.0	0
1	1000000002	5.0	1987.0	0	1.0	2.600789e+09	31.0	released	33.0	0
2	1000000003	6.0	1964.0	0	2.0	2.002000e+09	17.0	released	56.0	0
3	1000000004	7.0	1991.0	0	1.0	2.600789e+09	9.0	released	29.0	0
4	1000000005	9.0	1992.0	0	2.0	1.000000e+09	2.0	released	28.0	1

5 rows × 193 columns

Split the data into test and train subsamples

In [40]:

```
df.columns
```

Out[40]:

```
Index(['patient_id', 'global_num', 'birth_year', 'disease', 'infection_order',
       'infected_by', 'contact_number', 'state', 'n_age', 'sex_female',
       ...
       'infection_case_Pilgrimage to Israel',
       'infection_case_River of Grace Community Church',
       'infection_case_Seongdong-gu APT', 'infection_case_Shincheonji Church',
       'infection_case_Suyeong-gu Kindergarten',
       'infection_case_contact with patient', 'infection_case_etc',
       'infection_case_gym facility in Cheonan',
       'infection_case_gym facility in Sejong',
       'infection_case_overseas inflow'],
      dtype='object', length=193)
```

In [45]:

```
from sklearn.model_selection import train_test_split

# dont forget to define your X and y
X= df.drop(['state'],axis=1)
y=df['state']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, random_state=1)
X_train = pd.get_dummies(X_train)
X_test = pd.get_dummies(X_test)
```

Scale data to prep for model creation

In [46]:

```
#scale data
from sklearn import preprocessing
import numpy as np
# build scaler based on training data and apply it to test data to then also scale the te
st data
scaler = preprocessing.StandardScaler().fit(X_train)
X_train_scaled=scaler.transform(X_train)
X_test_scaled=scaler.transform(X_test)
```

In [47]:

```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import f1_score
from sklearn.metrics import auc
from sklearn.linear_model import LogisticRegression
from matplotlib import pyplot
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import f1_score
from sklearn.metrics import auc
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report,confusion_matrix,roc_curve,roc_auc_scor
e
from sklearn.metrics import accuracy_score,log_loss
from matplotlib import pyplot
```

Fit Random Forest Classifier

The fit model shows an overall accuracy of 80% which is great and indicates our model was effectively able to identify the status of a patients in the South Korea dataset.

In [48]:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=300, random_state = 1,n_jobs=-1)
model_res = clf.fit(X_train_scaled, y_train)
y_pred = model_res.predict(X_test_scaled)
y_pred_prob = model_res.predict_proba(X_test_scaled)
lr_probs = y_pred_prob[:,1]
ac = accuracy_score(y_test, y_pred)

f1 = f1_score(y_test, y_pred, average='weighted')
cm = confusion_matrix(y_test, y_pred)

print('Random Forest: Accuracy=%.3f' % (ac))
print('Random Forest: f1-score=%.3f' % (f1))
```

```
Random Forest: Accuracy=0.865
Random Forest: f1-score=0.832
```

Create Confusion Matrix Plots

Confusion matrices are great ways to review your model performance for a multi-class classification problem. Being able to identify which class the misclassified observations end up in is a great way to determine if you need to build additional features to improve your overall model. In the example below we plot a regular counts confusion matrix as well as a weighted percent confusion matrix. The percent confusion matrix is particulary helpful when you have unbalanced class sizes.

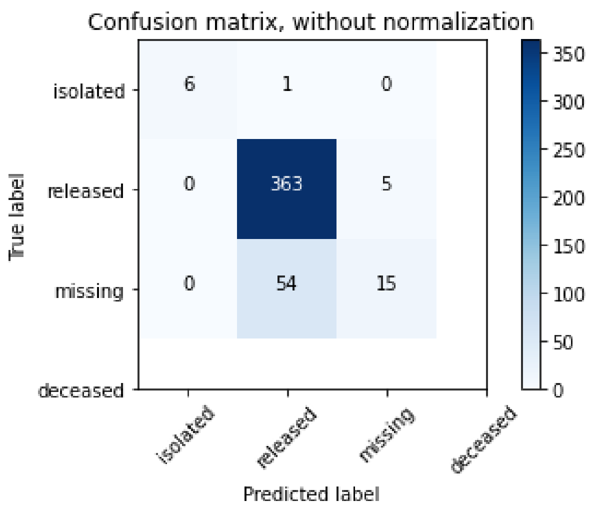
In [49]:

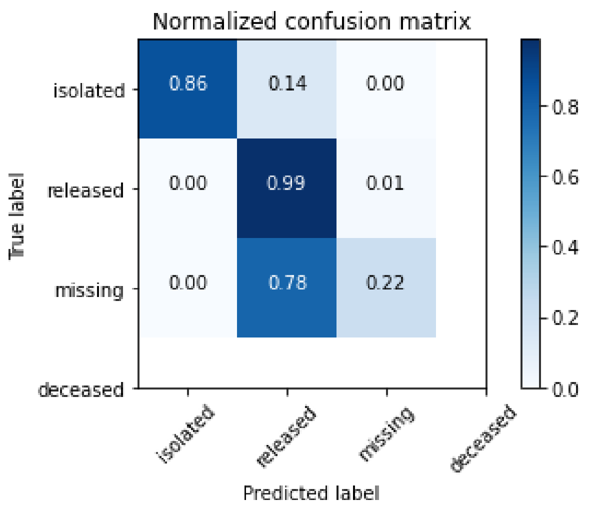
```
class_names=['isolated','released','missing','deceased'] # name  of classes
```

In [50]:

```
import itertools
import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()


# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                              title='Confusion matrix, without normalization')
#plt.savefig('figures/RF_cm_multi_class.png')

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                              title='Normalized confusion matrix')
#plt.savefig('figures/RF_cm_proportion_multi_class.png', bbox_inches="tight")
plt.show()
```

```
Confusion matrix, without normalization
[[  6   1   0]
 [  0 363   5]
 [  0  54  15]]
Normalized confusion matrix
[[0.86 0.14 0.  ]
 [0.   0.99 0.01]
 [0.   0.78 0.22]]
```





Plot feature importances

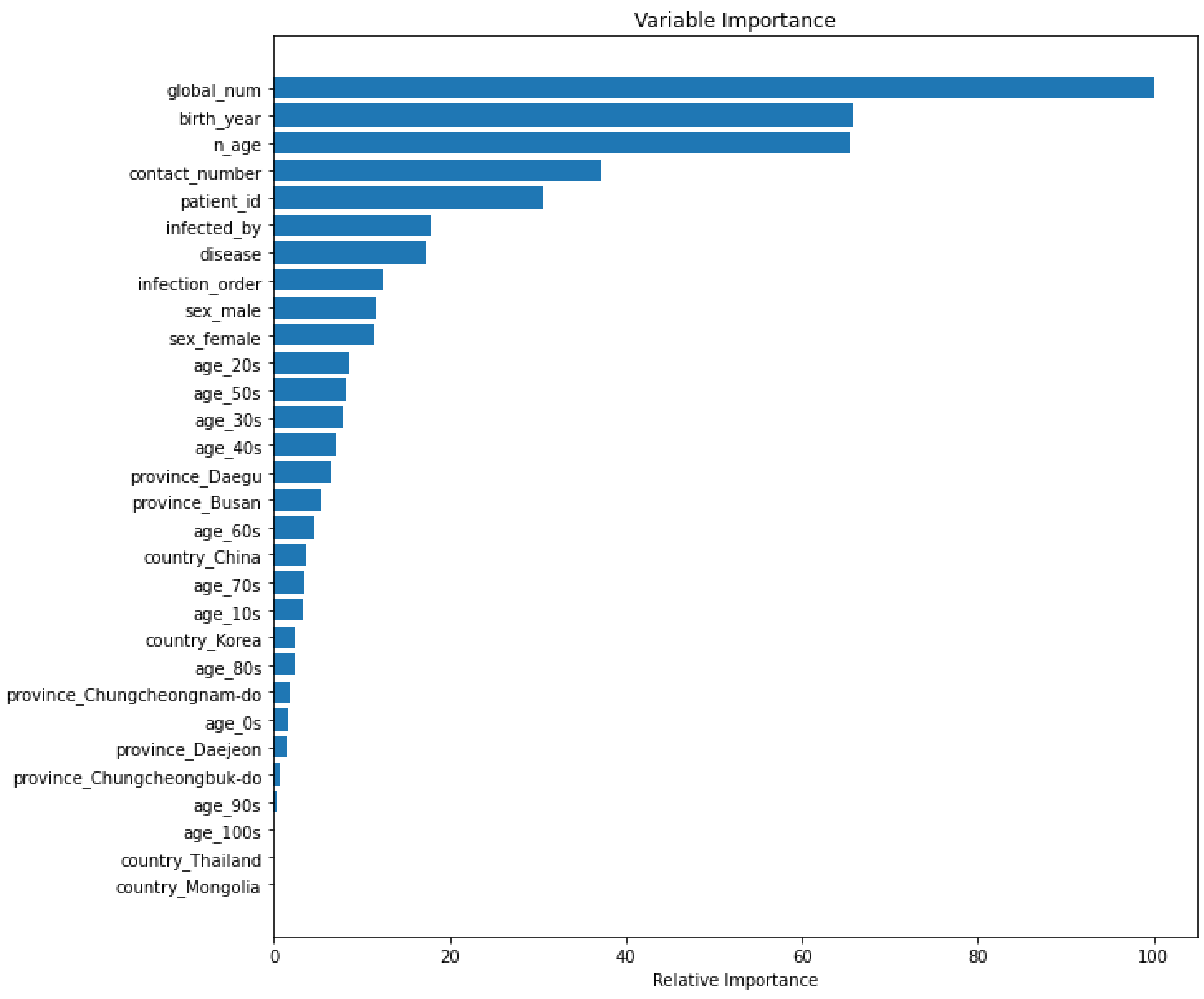
The random forest algorithm can be used as a regression or classification model. In either case it tends to be a bit of a black box, where understanding what's happening under the hood can be difficult. Plotting the feature importances is one way that you can gain a perspective on which features are driving the model predictions.

In [51]:

```
feature_importance = clf.feature_importances_
# make importances relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())[:30]
sorted_idx = np.argsort(feature_importance)[:30]

pos = np.arange(sorted_idx.shape[0]) + .5
print(pos.size)
sorted_idx.size
```

```
plt.figure(figsize=(10,10))
plt.barh(pos, feature_importance[sorted_idx], align='center')
plt.yticks(pos, X.columns[sorted_idx])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
plt.show()
```

```
30
```



The popularity of random forest is primarily due to how well it performs in a multitude of data situations. It tends to handle highly correlated features well, where as a linear regression model would not. In this case study we demonstrate the performance ability even with only a few features and almost all of them being highly correlated with each other. Random Forest is also used as an efficient way to investigate the importance of a set of features with a large data set. Consider random forest to be one of your first choices when building a decision tree, especially for multiclass classifications.