If all else fails and you've given up on rectifying your nans, you can always remove the sample or column completely, so it's no longer negatively impacting your analysis:

```
df = df.dropna(axis=0)  # row
df = df.dropna(axis=1)  # column

# Drop any row that has at least 4 NON-NaNs within it:
df = df.dropna(axis=0, thresh=4)
```

There may be cases where you want to get rid of non-nan values. For instance, if your dataset has a column you don't need:

```
# Axis=1 for columns
df = df.drop(labels=['Features', 'To', 'Delete'], axis=1)
```

You might also want to prune duplicate records if samples cannot have identical properties. Be careful though! To get rid of duplicate records, you should tell Pandas which features are to be examined, because Pandas generates indices for you automatically when you load a dataframe without specifying an index column. With each column having a unique index, Pandas won't find any 'duplicates' unless you limit your search to a subset of your dataframe's features:

```
df = df.drop_duplicates(subset=['Feature_1', 'Feature_2'])
```

Removing duplicate samples will cause gaps to occur in your index count. You can interpolate to fill those holes where appropriate, or alternatively you can reindex your dataframe:

```
df = df.reset_index(drop=True)
```

The `drop=True` parameter tells Pandas not to keep a backup copy of the original index. Most, if not all, of the above methods return a copy of your dataframe. This is useful because you can chain methods:

```
df = df.dropna(axis=0, thresh=2).drop(labels=['ColA', axis=1]).drop_duplicates(subset=['ColB', 'ColC']).reset_index()
```

However there may be times where you want these operations to work in-place on the dataframe calling them, rather than returning a new dataframe. Pass `inplace=True` as a parameter to any of the above methods to get that working.


Download on the
App Store
ANDROID APP ON
Google play