In [ ]:
```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebook_toc.js')
```

Application to CTA Data and Boosting

Table of Contents

The story so far:

- Linear Discriminant Analysis (LDA) and Fisher's linear discriminant
- Principal Component Analysis (PCA)
- Feature Selection
- Supervised Learning
- Clustering

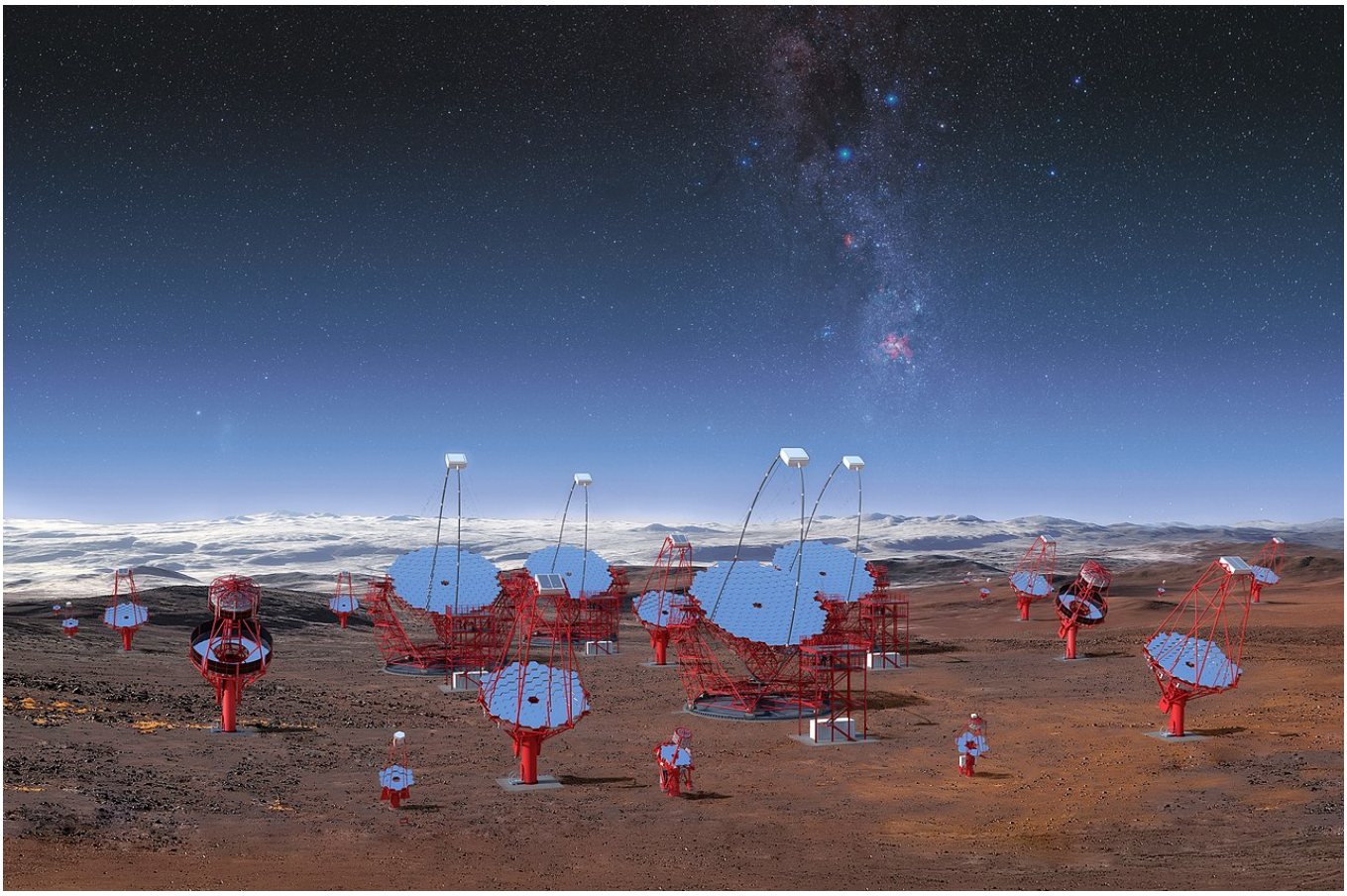
In [ ]:
```
from ml import plots
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from matplotlib.colors import ListedColormap
```

In [ ]:
```
pd.options.display.max_rows = 10
plots.set_plot_style()
```

In [ ]:
```
%matplotlib inline
```

A Complete Example

Below we load a dataset containing data from simulated CTA Observations.



We will perform the typical steps to build and evaluate a classifier.

0. Understand where your data comes from

1. Preprocessing

- Drop Constant Values,
- Handle Missing Data
- Feature Generation

2. Splitting

- Split your data into training and evaluation sets

3. Training

- Train your classifier of choice.

4. Evaluation

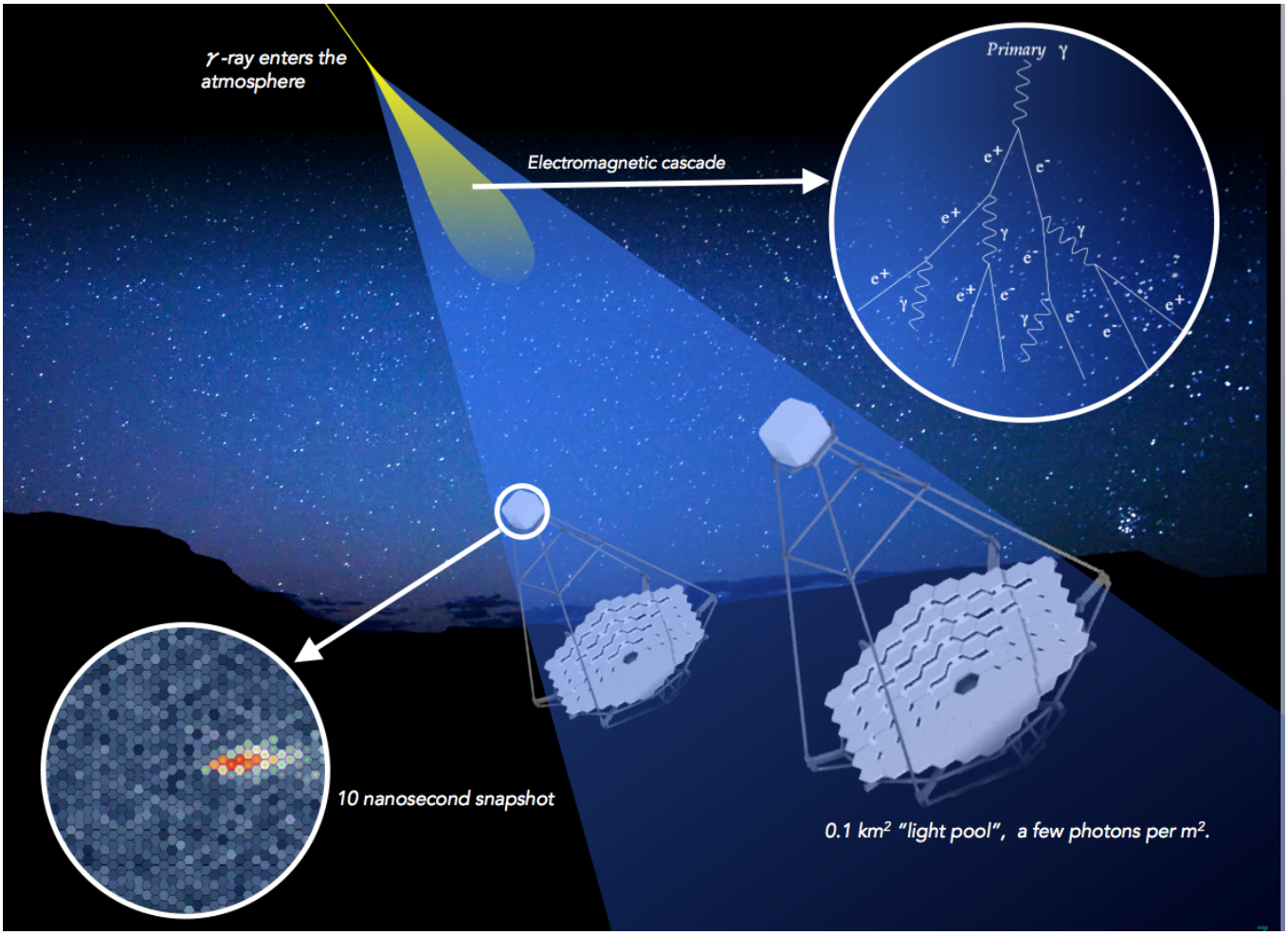
- Evaluate the performance on the test data set.
- If not good enough, go back to step 1

5. Physics

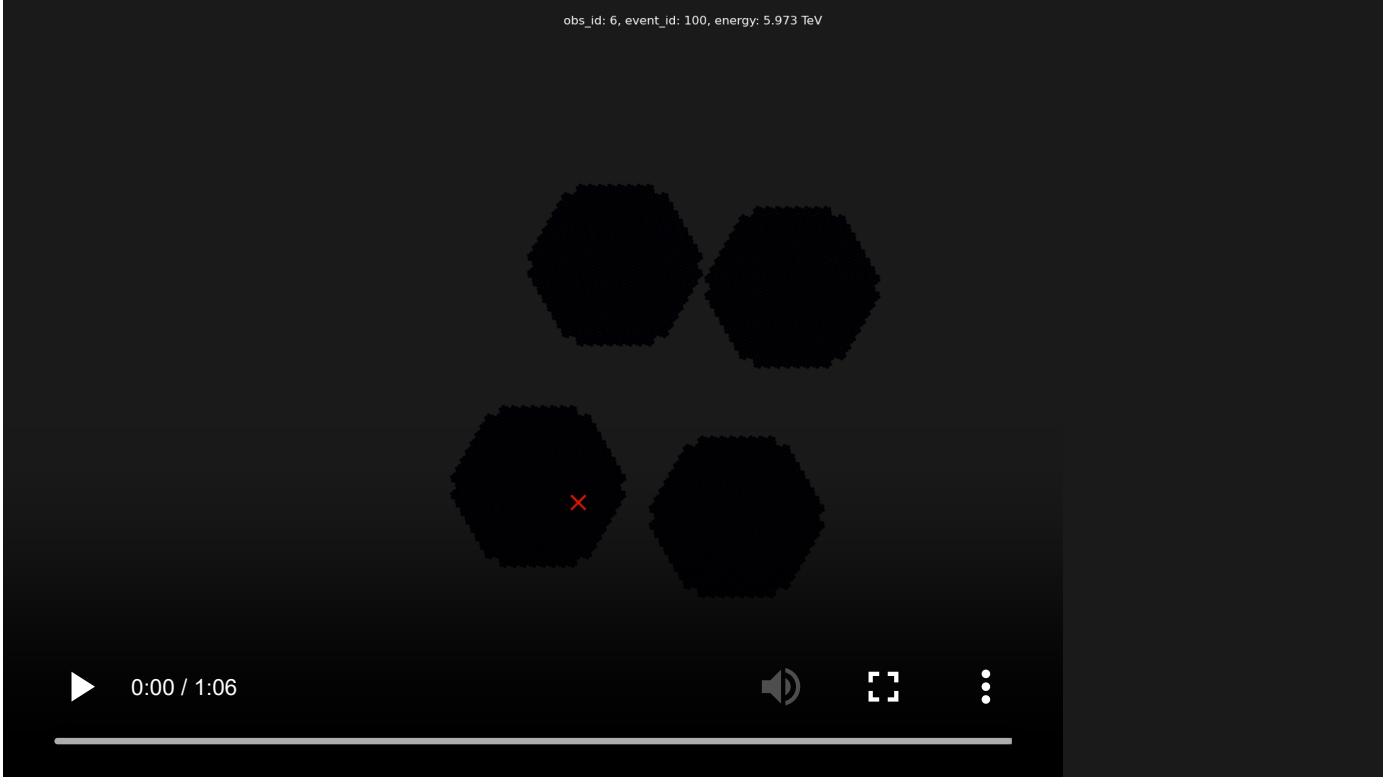
- Check whether your data support your hypothesis

1. Get to know your data

Cherenkov telescopes record short flashes of light produced by very high energy cosmic rays and photons hitting earths atmosphere.



In [ ]:
```
%%HTML
<!-- https://nextcloud.e5.physik.tu-dortmund.de/index.php/s/e7yb2mifGDeyDBN/download -->
<video width="100%" controls>
  <source src="./resources/lstsubarray_stereo.mp4" type="video/mp4">
</video>
```



We will use machine learning for two tasks in this example.

- Train a classifier to distinguish events induced by gamma rays form events induced by cosmic rays
- Train a regressor to estimate the energy of the incoming primary particle.

2. Preprocess data

A ***lot*** of preprocessing has *already* happened at this point.

- Calibration of Raw Data
- Data Reduction from voltage timeseries per pixel to number of photons and mean time for each pixel
- Calculation of image features
- Geometrical Reconstruction of the Shower Geometry

Load data and remove unwanted columns and store the true labels separately.

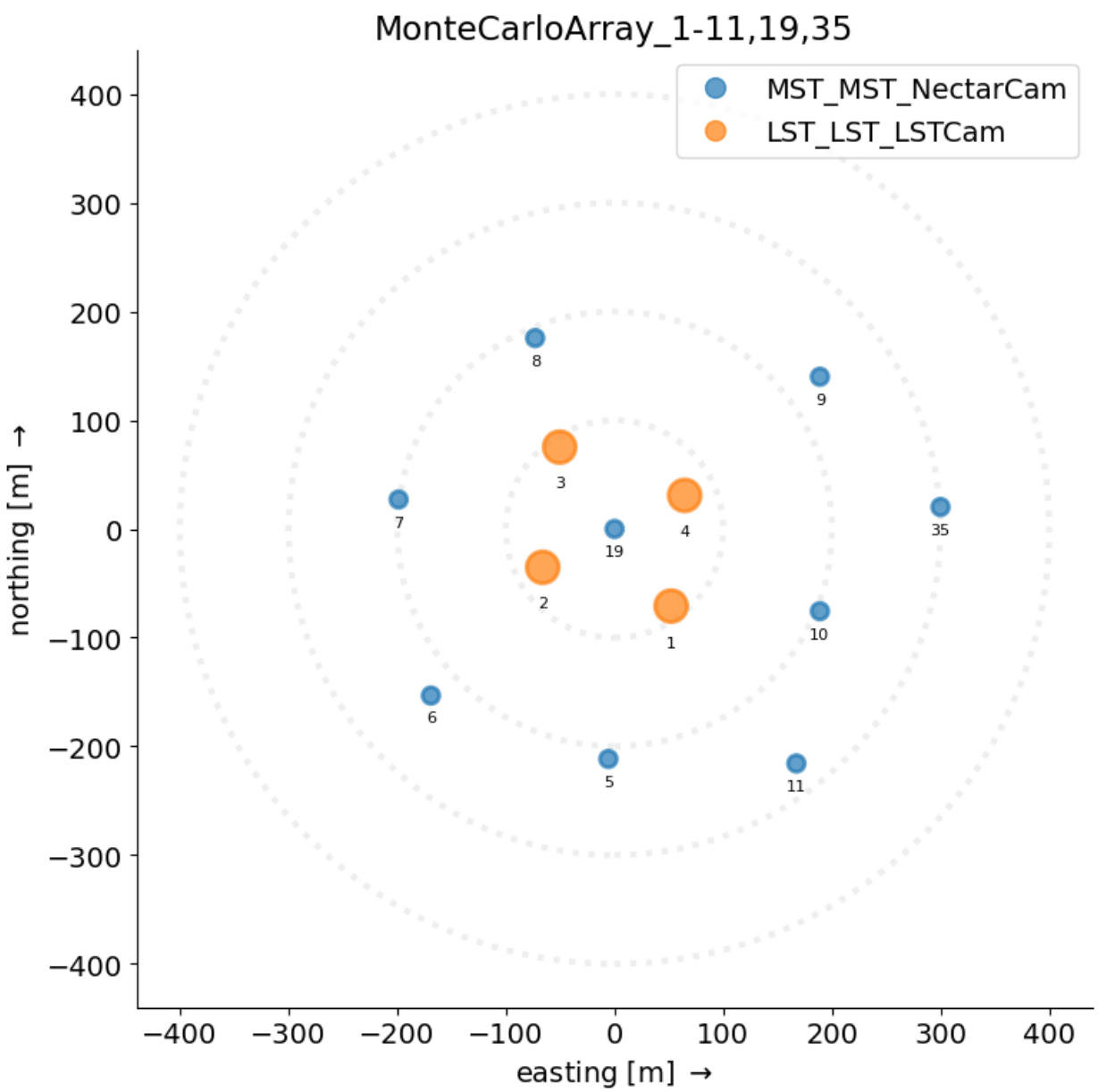
In [ ]:
```
import pandas as pd
from ctapipe.io import TableLoader
from ctapipe.utils import get_dataset_path
```

The dataset here is very similar but much smaller than the full dataset released publicly here: DOI 10.5281/zenodo.7298569

In [ ]:
```
gamma_path = get_dataset_path('gamma_diffuse_dl2_train_small.dl2.h5')
proton_path = get_dataset_path('proton_dl2_train_small.dl2.h5')
```

In [ ]:
```
with TableLoader(gamma_path) as loader:
    subarray = loader.subarray

subarray.peek()
```

```
/home/renlephy/miniconda3/envs/ml/lib/python3.10/site-packages/ctapipe/instrument/subarray.py:395: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
```



In [ ]:
```
def read_events(path):

    loader = TableLoader(
        path,
        load_dl2=True,
        load_instrument=True,
        load_simulated=True,
    )

    table = loader.read_telescope_events()

    # these two columns are arrays in each row, which is not supported by pandas
    table.remove_columns(['tels_with_trigger', 'HillasReconstructor_telescopes'])

    # convert astropy.table.Table to pd.DataFrame
    return table.to_pandas()
```

In [ ]:
```
gammas = read_events(gamma_path)
```

In [ ]:
```
len(gammas.columns)
```

Out[ ]:
```
92
```

Now delete all simulated values which can not be observed during measurement in the physical world. We know which columns to remove because they have a special prefix.

In [ ]:
```
forbidden_columns = 'true_|obs_id|event_id'
gammas = gammas.filter(regex=f'^(?!{forbidden_columns}).*$')

len(gammas.columns)
```

Out[ ]:
```
80
```

Check the data types of the columns. We can select non-numeric types and encode them. But in this case we might as well drop them as the attribute is not important.

In [ ]:
```
c = gammas.select_dtypes(exclude=['number', 'bool']).columns
print('Removed columns:', c.values)

gammas = gammas.drop(c, axis='columns')
```

```
Removed columns: ['time_mono' 'name' 'type' 'camera_name' 'optics_name_1' 'tel_description'
 'optics_name_2' 'size_type' 'reflector_shape' 'time']
```

We can spot the columns with constant values by looking at the count and/or standard deviation.

In [ ]:
```
desc = gammas.describe()
desc
```

Out[ ]:

	tel_id	n_trigger_pixels	hillas_intensity	hillas_skewness	hillas_kurtosis	hillas_fov_lon	hillas_fov_lat	hillas_r	
count	94572.000000	94572.000000	71529.000000	71529.000000	71529.000000	71529.000000	71529.000000	71529.000000	7152
mean	9.662257	19.123483	1093.364843	0.001066	2.348637	0.015764	0.026184	2.336999	
std	8.854069	33.652931	8851.418858	0.456087	0.759267	1.800013	1.787567	0.987287	10
min	1.000000	-1.000000	50.003008	-5.572948	1.003440	-3.739235	-4.054492	0.010258	-17
25%	4.000000	-1.000000	96.439808	-0.259209	1.942667	-1.309211	-1.322928	1.579093	-8
50%	7.000000	15.000000	194.843542	-0.000092	2.201602	0.016291	0.047204	2.257430	
75%	10.000000	22.000000	503.490359	0.263246	2.557658	1.360622	1.368379	3.249717	9
max	35.000000	1694.000000	832645.208949	3.365842	33.352201	3.745594	4.064763	4.080192	17

8 rows × 69 columns

In [ ]:
```
c = desc.columns[desc.loc['std'] == 0]
print('Removed columns:', c.values)
gammas = gammas.drop(c, axis='columns')
```

```
Removed columns: ['n_mirrors' 'event_type']
```

drop columns where all rows are nan

In [ ]:
```
c = gammas.columns[gammas.count() == 0]
print('Removed columns:', c.values)
gammas = gammas.drop(c, axis='columns')
```

```
Removed columns: ['HillasReconstructor_tel_impact_distance_uncert'
 'HillasReconstructor_core_uncert_x' 'HillasReconstructor_core_uncert_y'
 'HillasReconstructor_core_tilted_uncert_x'
 'HillasReconstructor_core_tilted_uncert_y'
 'HillasReconstructor_h_max_uncert' 'HillasReconstructor_goodness_of_fit']
```

here we do a specific pre-selection, again using "expert knowledge"

In [ ]:
```
print(len(gammas))
gammas = gammas[gammas['hillas_width'] > 0]
print(len(gammas))
```

```
94572
71393
```

Check for missing data. (Just delete it in this case)

In [ ]:
```
print(len(gammas))
gammas = gammas.dropna()
print(len(gammas))
```

```
71393
47149
```

So far we only loaded simulated gamma-ray showers. Now we do the same for the cosmic ray events. We create a method to perform all preprocessing in one step. We need this several times.

In [ ]:
```
def preprocess(df):
    df = df.filter(regex=f'^(?!{forbidden_columns}).*$')

    c = df.select_dtypes(exclude=['number', 'bool']).columns
    df = df.drop(c, axis='columns')
```

```
    c = df.columns[df.count() == 0]
    df = df.drop(c, axis='columns')

    desc = df.describe()

    c = desc.columns[desc.loc['std'] == 0]
    df = df.drop(c, axis='columns')

    df = df[df['hillas_width'] > 0]

    df = df.dropna()

    return df
```

In [ ]:
```
gammas = read_events(gamma_path)
gammas = preprocess(gammas)

protons = read_events(proton_path)
protons = preprocess(protons)
```

Now we can perform feature generation. We use our expert knowledge or intuition to create a new feature by combining existing columns into a new variable.

In [ ]:
```
def feature_generation(df):
    df['awesome_feature'] =  df.eval('hillas_intensity / (hillas_width * hillas_length)')

    # distance of impact point to the telescope
    df['impact'] = np.sqrt(
        (df['HillasReconstructor_core_x'] - df['pos_x'])**2
        + (df['HillasReconstructor_core_y'] - df['pos_y'])**2
    )

    return df

gammas = feature_generation(gammas)
protons = feature_generation(protons)

gammas[['awesome_feature', 'impact']]
```

Out[ ]:

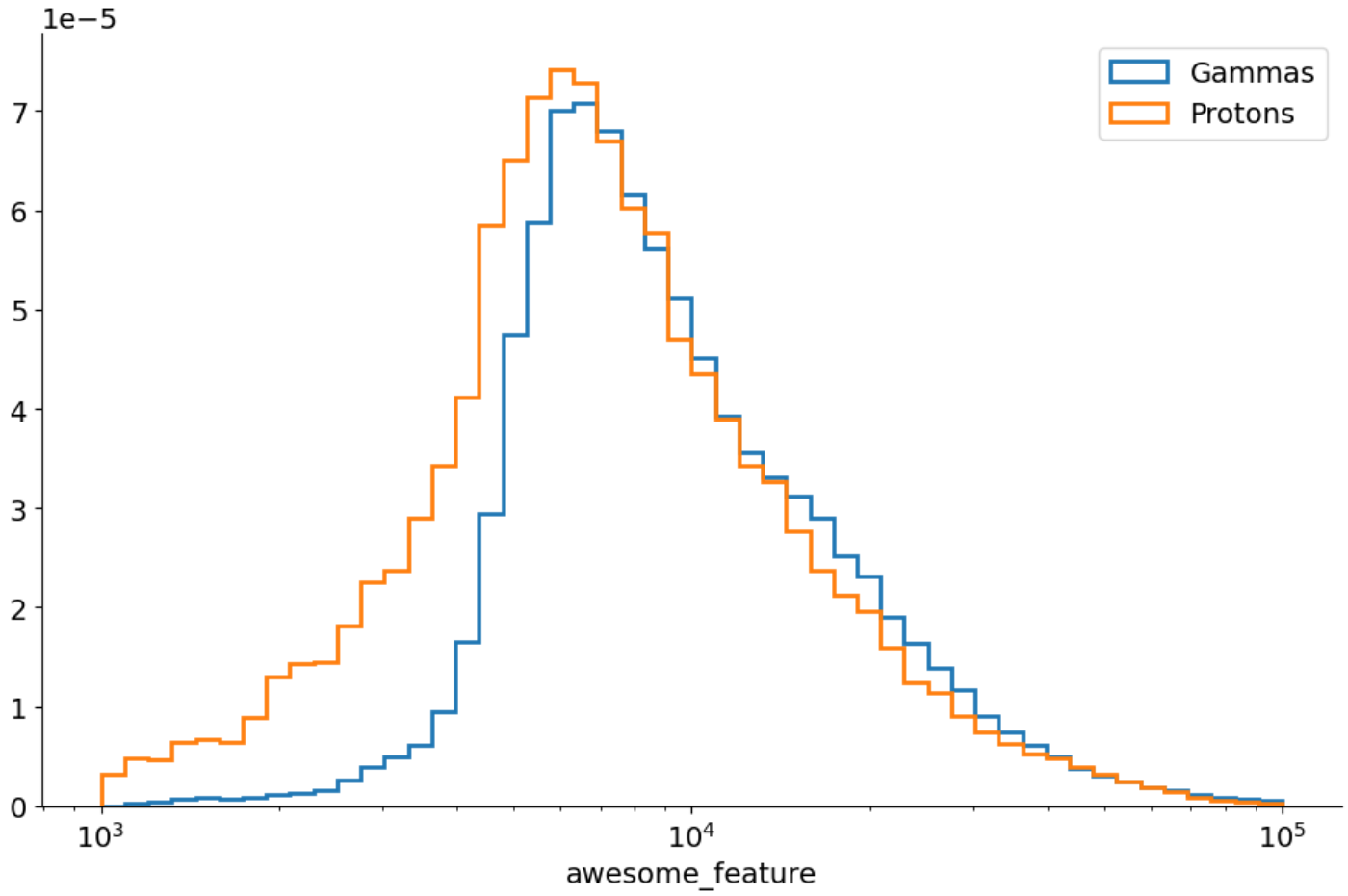
	awesome_feature	impact
1	12765.510477	169.127244
2	11649.969682	111.919552
3	18331.851001	68.077476
10	24262.177386	228.334539
12	9574.692367	138.256186
...	...	...
94563	12764.323411	182.768460
94564	8614.788850	164.233945
94569	10564.909904	123.754880
94570	10059.423221	70.801289
94571	9873.274675	169.854538

47149 rows × 2 columns

A quick look at the data so far

In [ ]:
```
# bins = np.geomspace(0.01, 1, 101)
# bins = np.logspace(0, 1, 100)
# bins = 100
# bins = np.arange(0, 10) - 0.5
bins = np.geomspace(1e3, 1e5, 51)

col = 'awesome_feature'

plt.figure()
plt.hist(gammas[col], bins=bins, histtype='step', lw=2, label='Gammas', density=True)
plt.hist(protons[col], bins=bins, histtype='step', lw=2, label='Protons', density=True)

plt.xscale('log')
```

```
plt.xlabel(col)
plt.legend()
None
```



At this point we combine the two datasets into one big matrix and build a label vector y

In [ ]:
```
X = pd.concat([gammas, protons])
y = np.concatenate([np.ones(len(gammas)), np.zeros(len(protons))])
```

3. Split Data

Now we can split the data into test and training sets. Scikit-Learn provides some neat methods to do just that.

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

X_test, X_train, y_test, y_train = train_test_split(X, y)
```

4. Train the classifier

Now we can train any classifier we want on the prepared data.

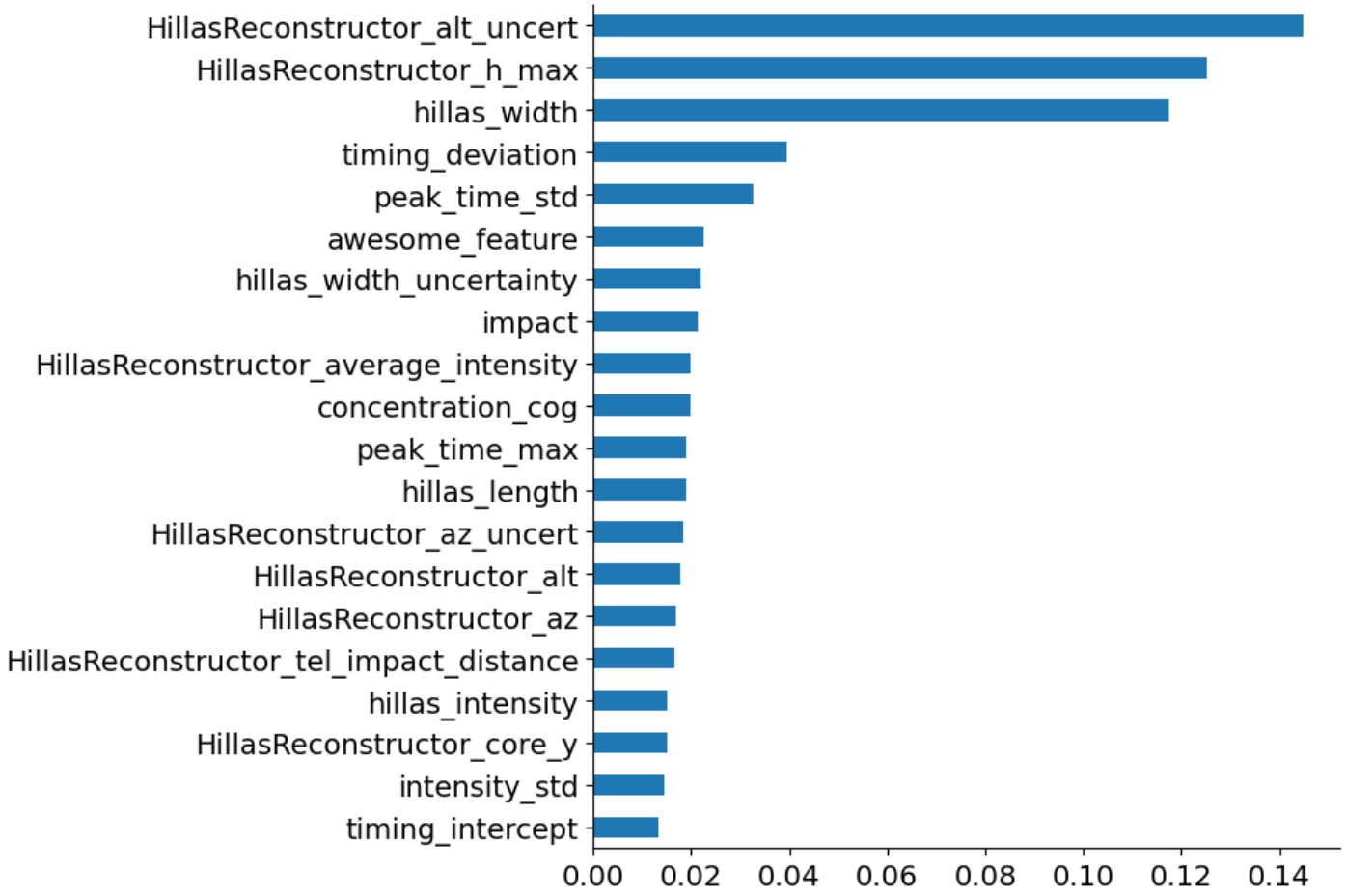
In [ ]:
```
from sklearn.tree import DecisionTreeClassifier

rf = DecisionTreeClassifier(max_depth=15, criterion='entropy')
rf.fit(X_train, y_train)

y_prediction = rf.predict(X_test)
y_prediction_proba = rf.predict_proba(X_test)
```

5. Evaluation

Check accuracy of the models and other metrics

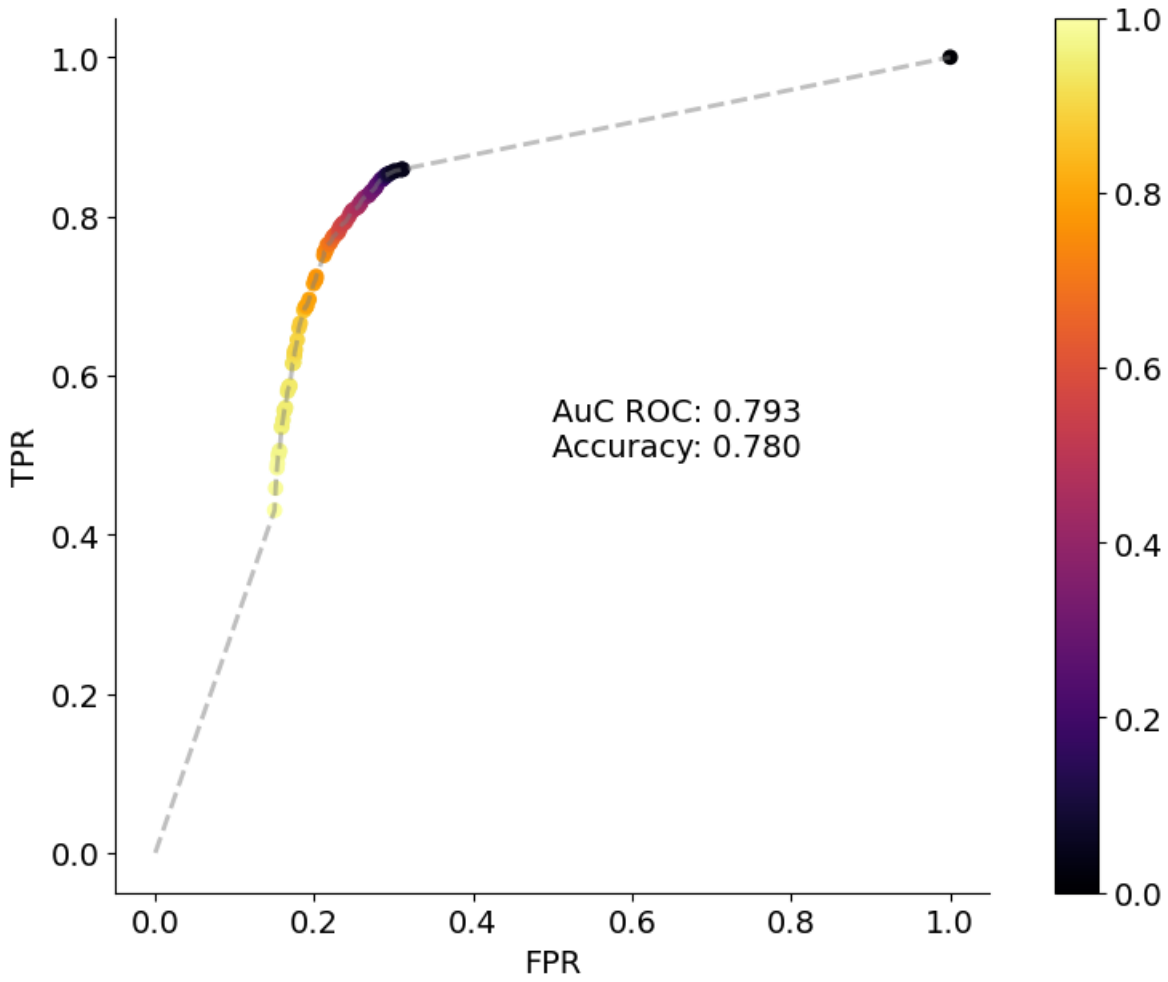
In [ ]:
```
importance = pd.Series(rf.feature_importances_, index=gammas.columns)


plt.figure()
importance.sort_values().tail(20).plot.barh()
```

Out[ ]:
```
<Axes: >
```



In [ ]:
```
from sklearn.metrics import accuracy_score, roc_curve, roc_auc_score

acc = accuracy_score(y_test, y_prediction)
auc = roc_auc_score(y_test, y_prediction_proba[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_prediction_proba[:, 1])
```

In [ ]:
```
def plot_roc(fpr, tpr, thresholds):
    fig, ax = plt.subplots()

    ax.plot(fpr, tpr, '--', color='gray', alpha=0.5)
    plot = ax.scatter(fpr, tpr, c=thresholds, vmax=1)
    fig.colorbar(plot)
    ax.text(0.5, 0.5, f'AuC ROC: {auc:0.03f} \nAccuracy: {acc:0.03f}')


    ax.set_xlabel('FPR')
    ax.set_ylabel('TPR')
    ax.set_aspect(1)


plot_roc(fpr, tpr, thresholds)
None
```



Perform steps 3, 4, and 5 in one step using cross validation

In [ ]:
```
from sklearn.model_selection import cross_validate

rf = DecisionTreeClassifier(max_depth=12, criterion='entropy')

scoring = {'acc': 'accuracy',
           'auc': 'roc_auc',
           'recall': 'recall'}

results = cross_validate(rf, X, y, cv=5, scoring=scoring, return_train_score=True)
results
```

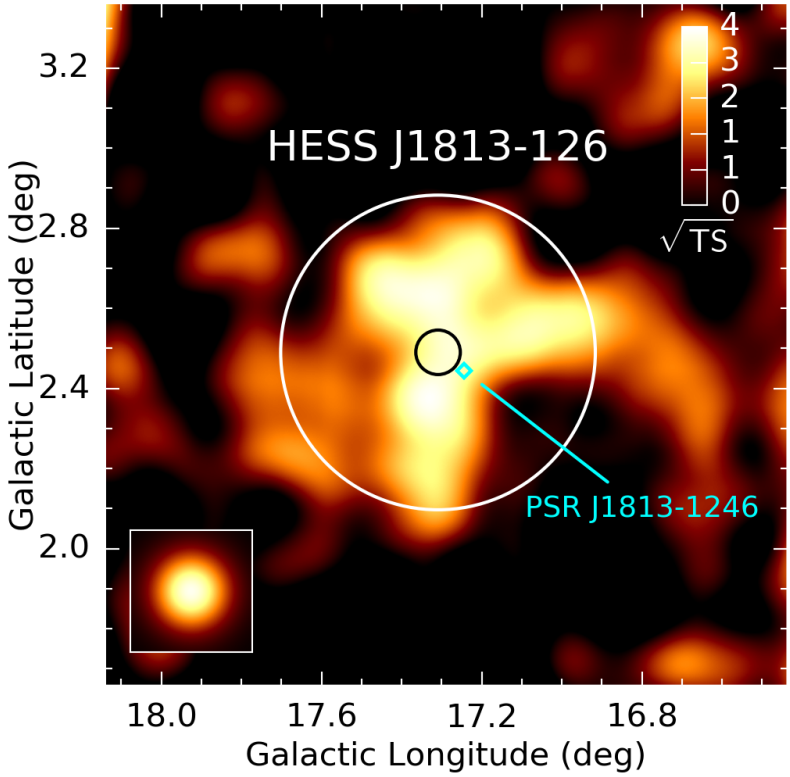
Out[ ]:
```
{'fit_time': array([3.10412216, 3.08952999, 3.07148409, 3.11081982, 3.06468964]),
 'score_time': array([0.01357889, 0.0143292 , 0.01342607, 0.01360655, 0.01364422]),
 'test_acc': array([0.79576341, 0.80946355, 0.78975459, 0.80758226, 0.80004402]),
 'train_acc': array([0.86974703, 0.87141148, 0.86970756, 0.87391675, 0.87592506]),
 'test_auc': array([0.84577459, 0.85444274, 0.83984564, 0.85154491, 0.84734676]),
 'train_auc': array([0.94453129, 0.94598571, 0.94356005, 0.94855694, 0.94816778]),
 'test_recall': array([0.86065748, 0.852386  , 0.83223754, 0.83478261, 0.85141584]),
 'train_recall': array([0.92656221, 0.90893184, 0.91518863, 0.91102627, 0.91680806])}
```

In [ ]:
```
auc = results['test_auc']
recall = results['test_recall']
acc = results['test_acc']

print(f'Area under RoC curve: {auc.mean():0.04f} ± {auc.std():0.04f}')
print(f'Accuracy:             {acc.mean():0.04f} ± {acc.std():0.04f}')
print(f'Recall:               {recall.mean():0.04f} ± {recall.std():0.04f}')
```

```
Area under RoC curve: 0.8478 ± 0.0050
Accuracy:             0.8005 ± 0.0073
Recall:               0.8463 ± 0.0110
```

6. Physics

Now we could test our model and our hypothesis on real observed data. This part of the analysis is the most time consuming in general. It also requires more data than than this notebook can handle. After careful analysis one can produce an image of the gamma-ray sky



Improving Classification

Boosting and AdaBoost

Similar to the idea of combining many classifiers through bagging (like we did for the RandomForests) we now train many estimators in a sequential manner. In each iteration the data gets modified slightly using weights w for each sample in the training data. In the first iteration the weights are all set to $w = 1$

In each successive iteration the weights are updated. The samples that were incorrectly classified in the previous iteration get a higher weight. The weights for correctly classified samples get decreases. In other words: We increase the influence/importance of samples that are difficult to classify.

Predictions are performed by taking a weighted average of the single predictors.

The popular AdaBoost algorithms takes this a step further by optimizing the weight of each separate classifier in the ensemble. The AdaBoost ensemble combines many learners in an iterative way. The learner at iteration m is:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

The choice of F_0 is problem specific.

Each weak learner produces a prediction $h(x_m)$ for each sample in the training set. At each iteration m a weak learner is fitted and assigned a coefficient γ_m which is found by minimizing:

$$\gamma_m = \arg\min_{\gamma} \sum_{i}^{N} E\big(F_{m-1}(x_i) + \gamma h(x_i)\big)$$

where $E(F)$ is some error function and x_i is the reweighted data sample.

In general this method can work with any classifying method. Traditionally it is being used with very small decision trees. The weights get used to select the split points during the minimization of the loss function in each node

$$\underset{(X,s)\in \mathbf{X}\times S}{\arg\max}\, IG(X,Y) = \underset{(X,s)\in \mathbf{X}\times S}{\arg\max}\, (H(Y) - H(Y|\, X)).$$

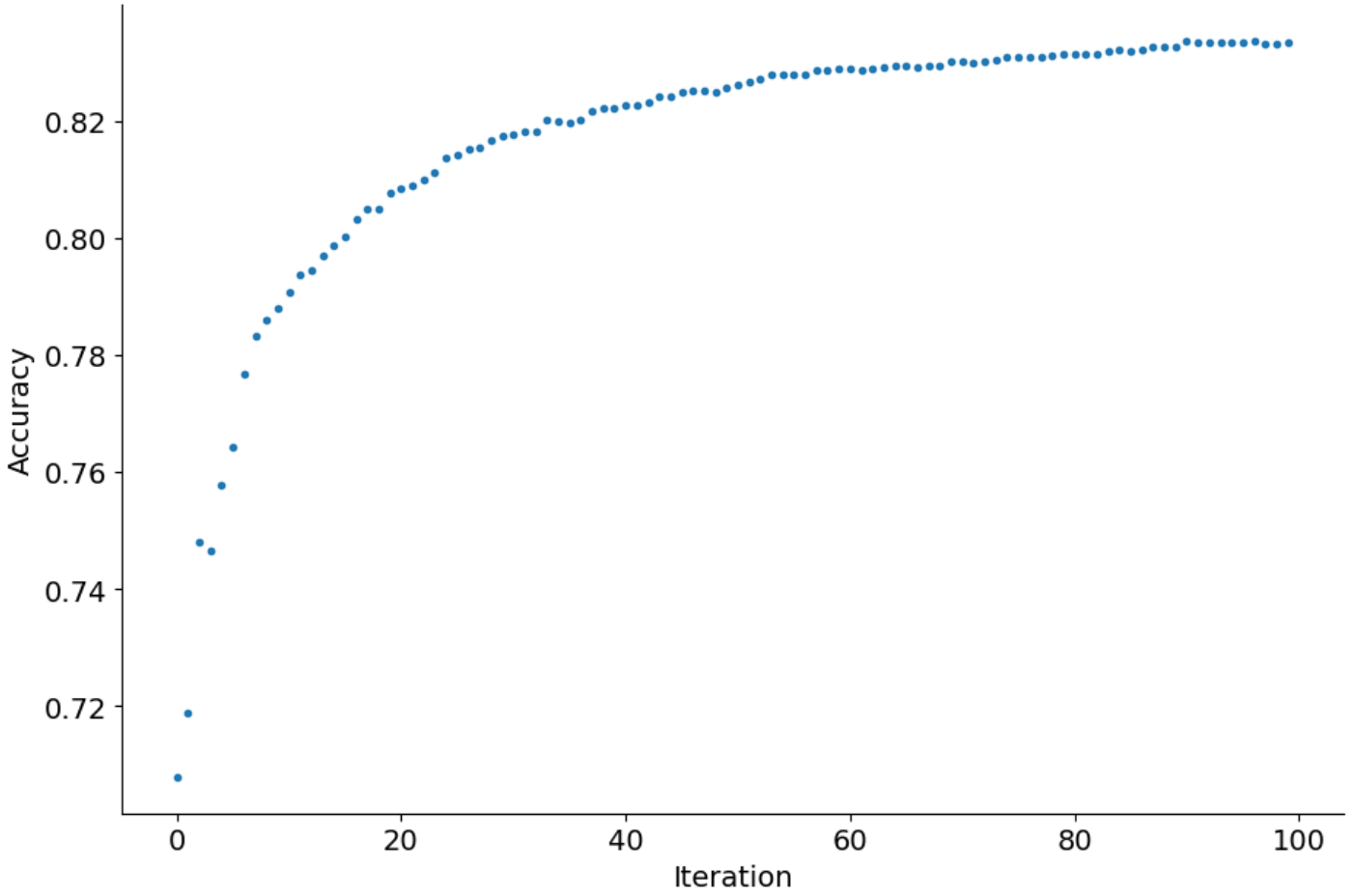
Below we try AdaBoost on the CTA data.

In [ ]:
```
from sklearn.ensemble import AdaBoostClassifier

ada = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=2),
    n_estimators=100,
    learning_rate=0.5,
)
ada.fit(X_train, y_train)
```

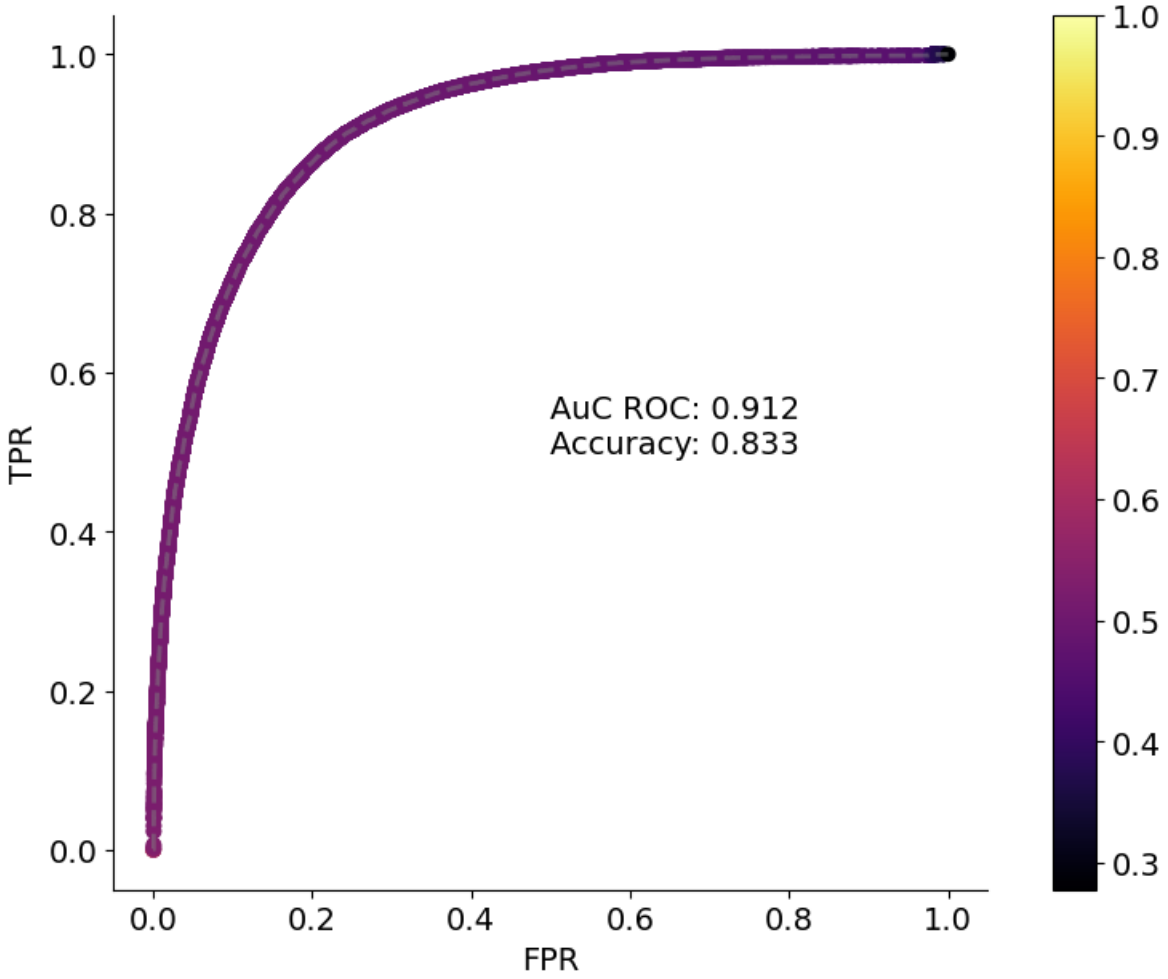
```
y_prediction = ada.predict(X_test)
y_prediction_proba = ada.predict_proba(X_test)
```

In [ ]:
```
scores = np.array(list(ada.staged_score(X_test, y_test)))

plt.figure()
plt.plot(scores, '.')
plt.ylabel('Accuracy')
plt.xlabel('Iteration')
None
```

```
/home/renlephy/miniconda3/envs/ml/lib/python3.10/site-packages/sklearn/base.py:464: UserWarning: X does not have valid fea
ture names, but AdaBoostClassifier was fitted with feature names
  warnings.warn(
/home/renlephy/miniconda3/envs/ml/lib/python3.10/site-packages/sklearn/base.py:464: UserWarning: X does not have valid fea
ture names, but AdaBoostClassifier was fitted with feature names
  warnings.warn(
```



In [ ]:
```
acc = accuracy_score(y_test, y_prediction)
auc = roc_auc_score(y_test, y_prediction_proba[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_prediction_proba[:, 1])

plot_roc(fpr, tpr, thresholds)
```



Gradient Boosting

Very similar to AdaBoost. Only this time we change the target label we train the classifiers for.

Formulate the general problem as follows (See Wikipedia):

Starts with a constant function $F_0(x)$ and some differentiable loss function L and incrementally expands it in a greedy fashion:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

$$F_m(x) = F_{m-1}(x) + \arg\min_{h_m \in \mathcal{H}} \left[\sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i)) \right]$$

Finding the best $h_m \in \mathcal{H}$ is computationally speaking impossible. If we could find the perfect h however, we know that

$$F_{m+1}(x_i) = F_m(x_i) + h(x_i) = y_i$$

or, equivalently,

$$h(x_i) = y_i - F_m(x_i)$$

Note that for the mean squared error loss $\frac{1}{2}(y_i - F(x_i))^2$ this is equivalent to the negative gradient with respect to F_i.

For a general loss function we fit $h_m(x)$ to the residuals, or negative gradients

$$r_{i,m} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

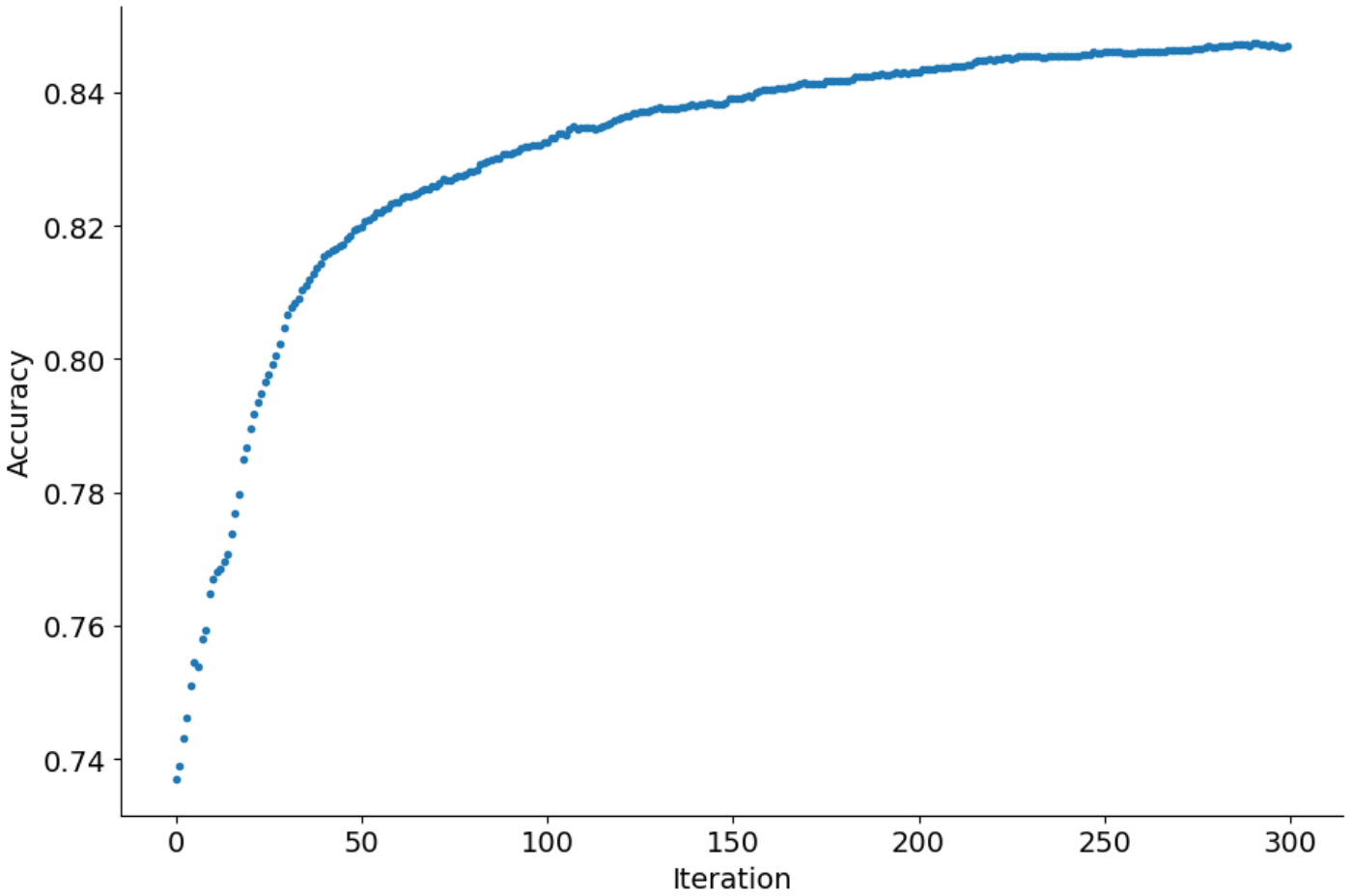
Below we try it on CTA data again.

In [ ]:
```
from sklearn.ensemble import GradientBoostingClassifier

grb = GradientBoostingClassifier(
    verbose=True,
    n_estimators=300,
)
grb.fit(X_train, y_train)
```

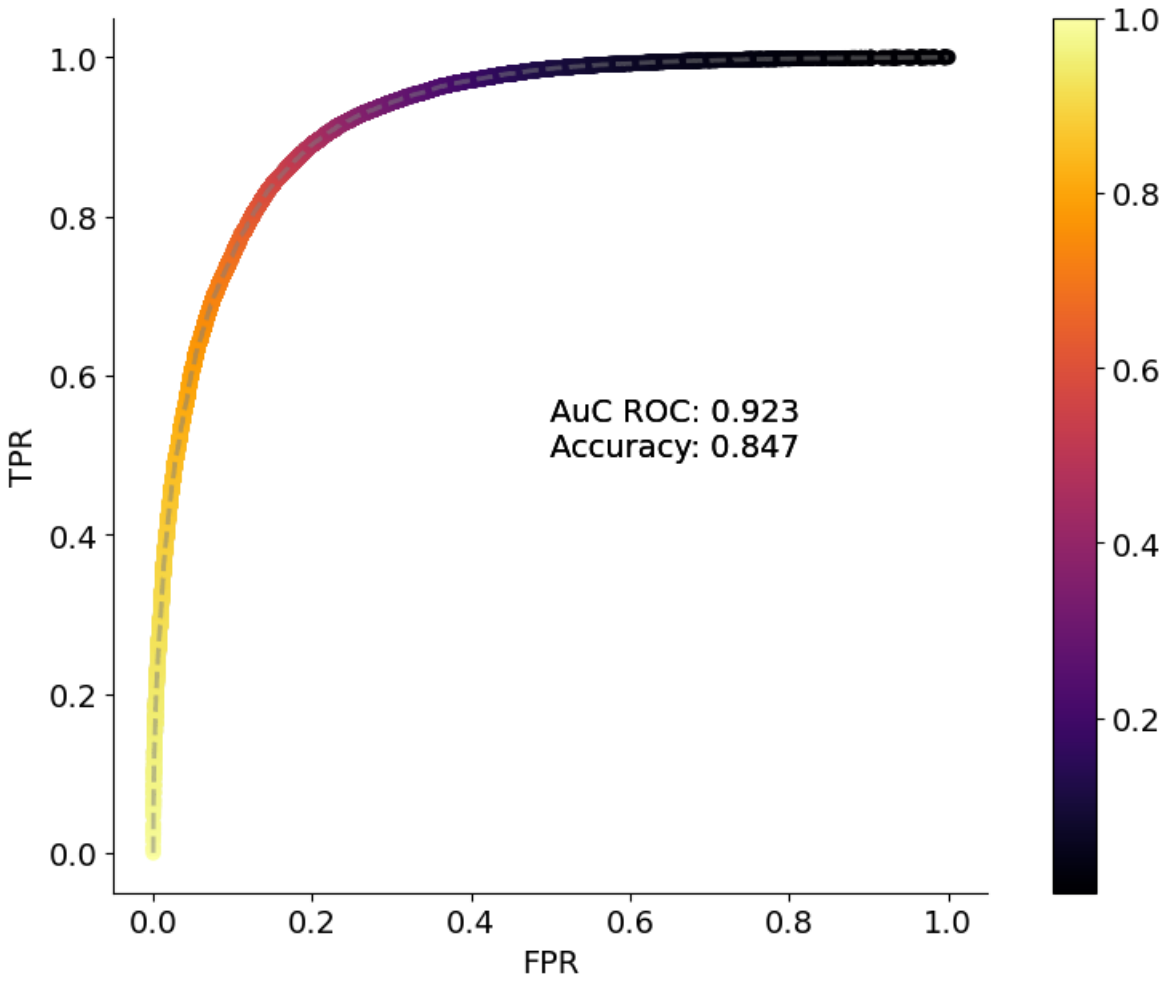
```
y_prediction = grb.predict(X_test)
y_prediction_proba = grb.predict_proba(X_test)
```

```
     Iter       Train Loss   Remaining Time
        1           1.3292            1.26m
        2           1.2826            1.26m
        3           1.2445            1.26m
        4           1.2122            1.26m
        5           1.1835            1.25m
        6           1.1566            1.25m
        7           1.1334            1.25m
        8           1.1120            1.24m
        9           1.0935            1.24m
       10           1.0771            1.24m
       20           0.9444            1.20m
       30           0.8746            1.16m
       40           0.8294            1.12m
       50           0.7980            1.07m
       60           0.7749            1.03m
       70           0.7569           59.36s
       80           0.7399           56.82s
       90           0.7257           54.26s
      100           0.7138           51.71s
      200           0.6355           25.87s
      300           0.5921            0.00s
```

In [ ]:
```
l = [accuracy_score(y_pred, y_test) for y_pred in grb.staged_predict(X_test)]

plt.figure()
plt.plot(range(len(l)), l, '.')
plt.ylabel('Accuracy')
plt.xlabel('Iteration')
None
```



In [ ]:
```
acc = accuracy_score(y_test, y_prediction)
auc = roc_auc_score(y_test, y_prediction_proba[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_prediction_proba[:, 1])

plot_roc(fpr, tpr, thresholds)

plt.text(0.5, 0.5, f'AuC ROC: {auc:0.03f} \nAccuracy: {acc:0.03f}')
None
```

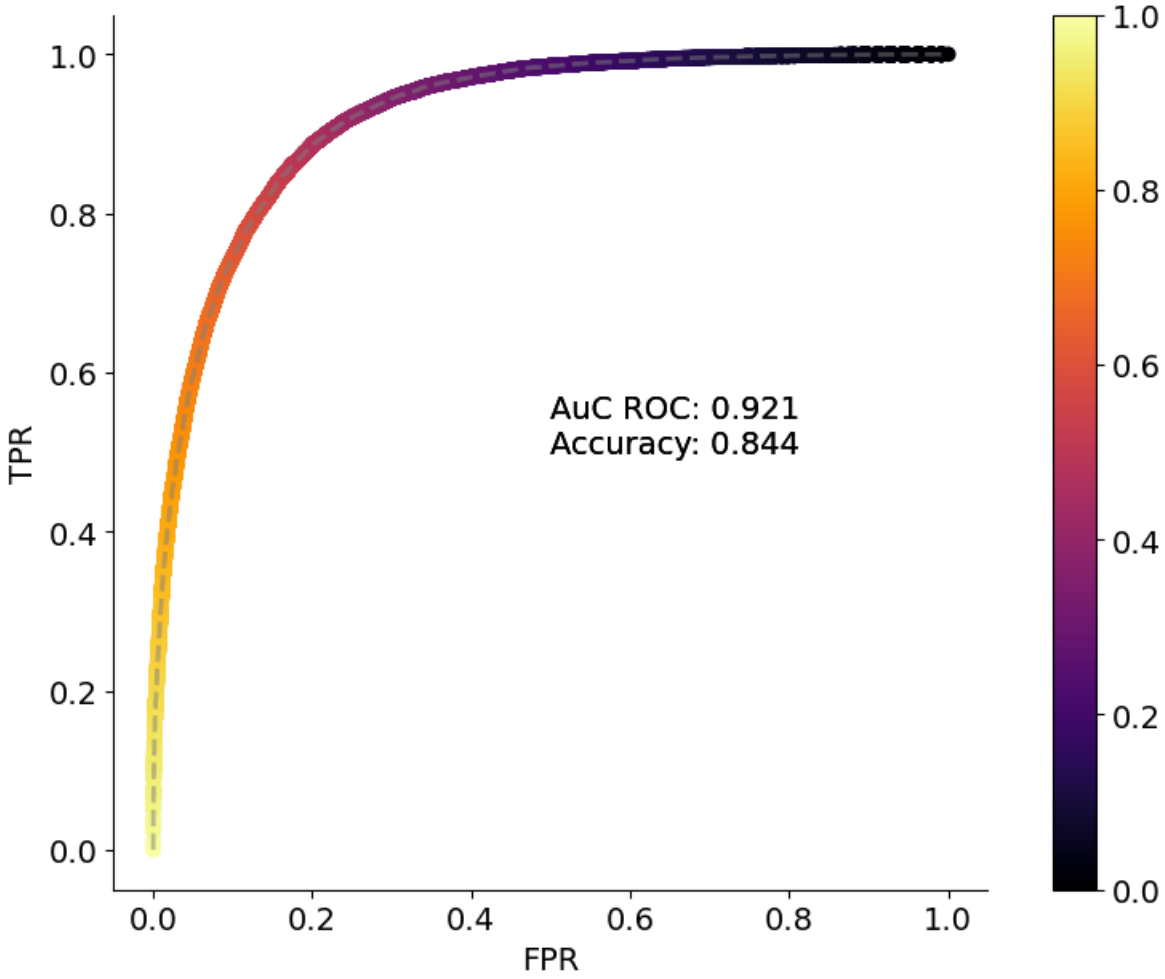


More on gradient descent algorithms can be found in the Neural Network lecture.

Let's now test our all time favorite classifier.

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=150,  max_depth=18, criterion='entropy')
rf.fit(X_train, y_train)

y_prediction = rf.predict(X_test)
y_prediction_proba = rf.predict_proba(X_test)
```

In [ ]:
```
acc = accuracy_score(y_test, y_prediction)
auc = roc_auc_score(y_test, y_prediction_proba[:, 1])
fpr, tpr, thresholds = roc_curve(y_test, y_prediction_proba[:, 1])

plot_roc(fpr, tpr, thresholds)
plt.text(0.5, 0.5, f'AuC ROC: {auc:0.03f} \nAccuracy: {acc:0.03f}')
None
```


TPR
FPR
AuC ROC: 0.921
Accuracy: 0.844
0.0
0.2
0.4
0.6
0.8
1.0