HW2

October 17, 2025

1 Problem 1

Radioactive decay of a nucleus is described as $N(t) = N_0 e^{-t/\tau}$ where N_0 is the number of the nucleus in question initially and τ is the half life of the nucleus. $N(t)$ is the number of nuclei remaining at time t.

1.1 Part 1

propagate the uncertainties on N_0 and τ to get the corresponding uncertainty on $N(t)$ analytically using the information discussed in class.

Type your answer below:

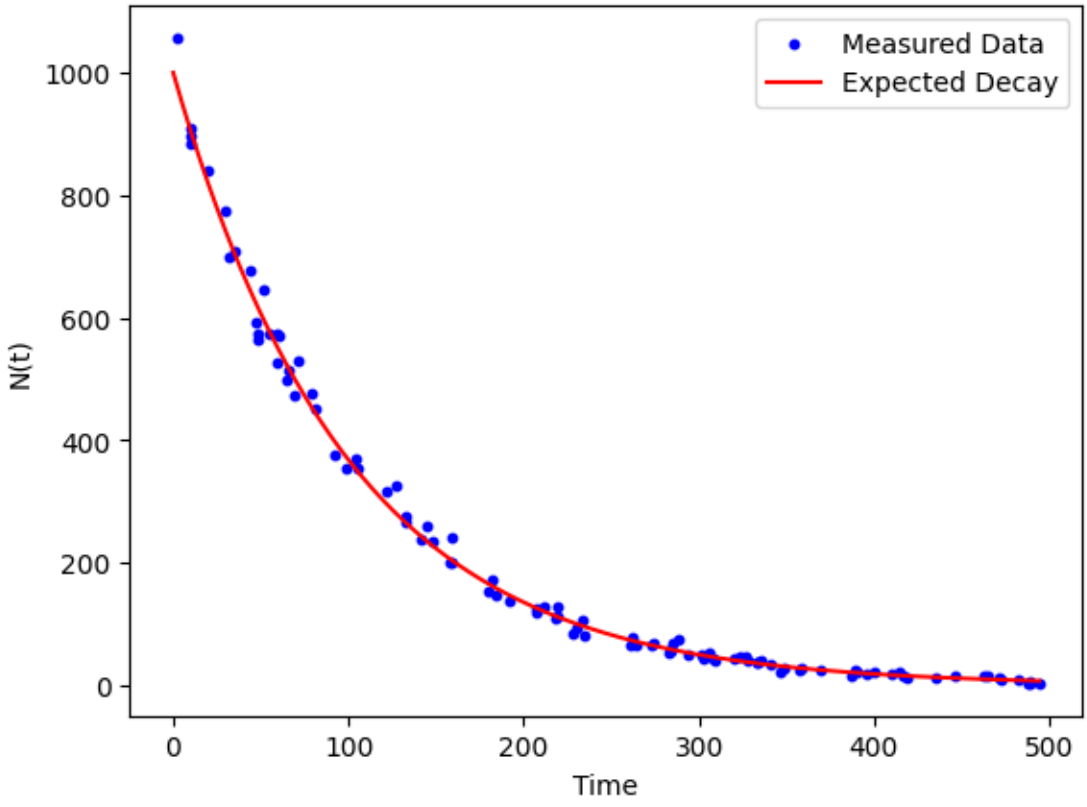
$$\sigma_{N(t)}^2 = \left(\frac{\partial N(t)}{\partial N_0}\right)^2 \sigma_{N_0}^2 + \left(\frac{\partial N(t)}{\partial \tau}\right)^2 \sigma_\tau^2$$
$$\sigma_{N(t)}^2 = \left(e^{-t/\tau}\right)^2 \sigma_{N_0}^2 + \left(N_0 e^{-t/\tau} \frac{t}{\tau^2}\right)^2 \sigma_\tau^2$$
$$\sigma_{N(t)}^2 = \left(e^{-t/\tau}\right)^2 \sigma_{N_0}^2 + \left(N(t) \frac{t}{\tau^2}\right)^2 \sigma_\tau^2$$
$$\sigma_{N(t)}^2 = \left(\frac{N(t)}{N_0}\right)^2 \sigma_{N_0}^2 + \left(\frac{tN(t)}{\tau^2}\right)^2 \sigma_\tau^2$$
$$\sigma_{N(t)}^2 = N(t)^2 \left[\left(\frac{\sigma_{N_0}}{N_0}\right)^2 + \left(\frac{t\sigma_\tau}{\tau^2}\right)^2\right]$$
$$\sigma_{N(t)} = N(t)\sqrt{\left(\frac{\sigma_{N_0}}{N_0}\right)^2 + \left(\frac{t\sigma_\tau}{\tau^2}\right)^2} = N_0 e^{-t/\tau}\sqrt{\left(\frac{\sigma_{N_0}}{N_0}\right)^2 + \left(\frac{t\sigma_\tau}{\tau^2}\right)^2}$$

1.2 Part 2

Write a program that draws random numbers according to what you would expect to measure in terms of $N(t)$ as a function of time for a given τ. Set N_0 to at least 1k, generate 100 samples for uniform random t-values in a reasonable range; samples at larger times will of course be more rare. For the plot purposes set $\tau = 100s$. Make a plot out to large enough t values.

[3]:
```
import numpy as np
import matplotlib.pyplot as plt

def simulate_decay(N0=1000, tau=100, n=100, t_max=500, seed=0):
    np.random.seed(seed)
    t_values = np.random.uniform(0, t_max, n)
    N_expected = N0 * np.exp(-t_values / tau)
    N_measured = np.random.poisson(N_expected)
```

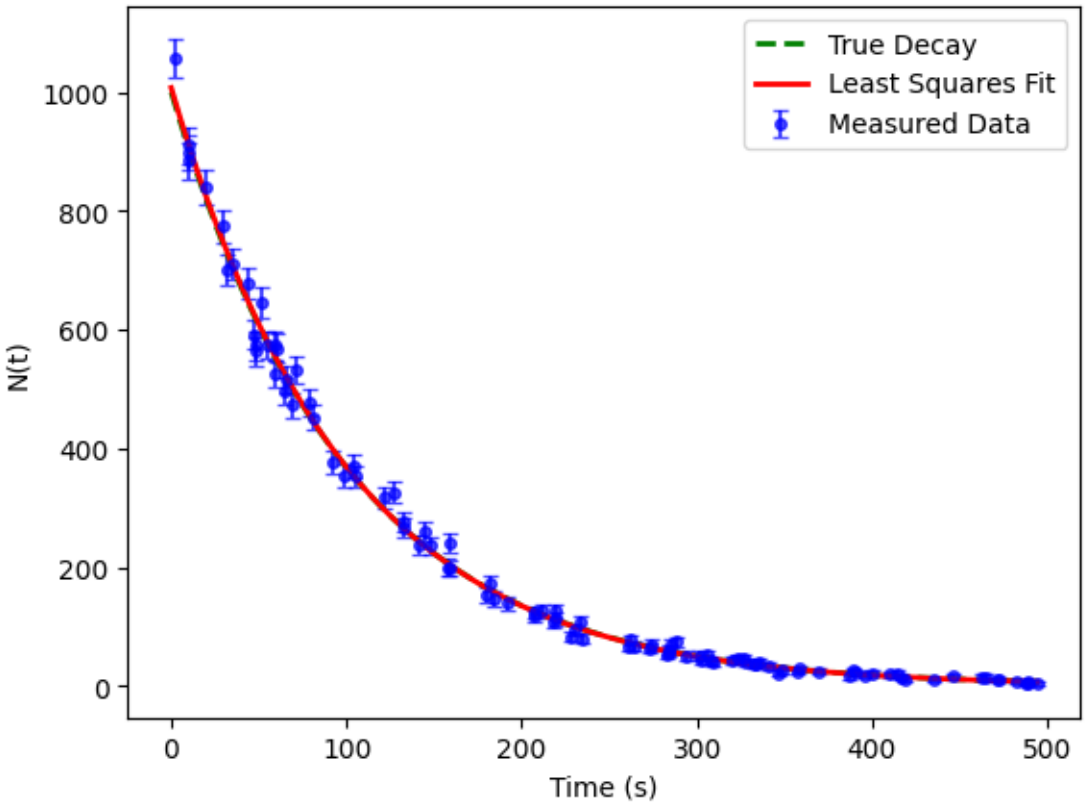
```
    sort_idx = np.argsort(t_values)
    t_sorted = t_values[sort_idx]
    N_measured_sorted = N_measured[sort_idx]

    return t_sorted, N_measured_sorted

N0 = 1000
tau = 100
n = 100
t, N_measured = simulate_decay(N0=N0, tau=tau, n=n)
plt.scatter(t, N_measured, label="Measured Data", color="blue", s=10)

t_smooth = np.linspace(0, max(t), 500)
N_expected_smooth = N0 * np.exp(-t_smooth / tau)
plt.plot(t_smooth, N_expected_smooth, label="Expected Decay", color="red")
plt.xlabel("Time")
plt.ylabel("N(t)")
plt.legend()
plt.show()
```



1.3 Part 3 (Bonus)

Fit $N(t)$ using a least squares fit to get τ and N_0. Make sure to keep track of the statistical uncertainties in your *data* and your *fit*. Do the values agree with those you set in *Part 2*? Are τ and $N(0)$ correlated with each other?

[7]:
```
from scipy.optimize import curve_fit

def decay_model(t, N0, tau):
    return N0 * np.exp(-t / tau)

uncertainties = np.sqrt(N_measured)
uncertainties[uncertainties == 0] = 1.0

p0 = [N0, tau]
popt, pcov = curve_fit(decay_model, t, N_measured, p0=p0,
                       sigma=uncertainties, absolute_sigma=True)

N0_fit, tau_fit = popt
N0_err, tau_err = np.sqrt(np.diag(pcov))

correlation = pcov[0,1] / (N0_err * tau_err)

residuals = N_measured - decay_model(t, *popt)
chi2 = np.sum((residuals / uncertainties)**2)
chi2_red = chi2 / (len(t) - 2)

plt.errorbar(t, N_measured, yerr=uncertainties, fmt="o",
             color="blue", alpha=0.7, label="Measured Data",
             markersize=4, capsize=3)

t_smooth = np.linspace(0, max(t), 500)
N_expected_smooth = N0 * np.exp(-t_smooth / tau)
plt.plot(t_smooth, N_expected_smooth, "--", color="green",
         linewidth=2, label=f"True Decay")

N_fit_smooth = decay_model(t_smooth, *popt)
plt.plot(t_smooth, N_fit_smooth, color="red", linewidth=2,
         label=f"Least Squares Fit")

plt.xlabel("Time (s)")
plt.ylabel("N(t)")
plt.legend()
plt.show()

print(N0_fit, N0_err)
print(tau_fit, tau_err)
print(correlation)
```




```
1007.1437643697402 9.620189360091334
99.46504026885202 0.7615739190836509
-0.7126024476535796
```

Fitted values for N_0 and τ are close to the values given in part 2. The correlation coefficient is -0.7, indicating a strong negative correlation between N_0 and τ.

2 Problem 2: Tabular data and BDTs: Classifying LHC collisions

2.0.1 Goal

Discriminate $H \to \tau\tau$ (signal) from background such as $t\bar{t}$.

In a real detector, the signal looks like:

2.0.2 Boosted decision tree library: XGBoost

We'll use the python API for the XGBoost (eXtreme Gradient Boosting) library.

2.0.3 Data

ATLAS hosted a Kaggle competition for identifying $H \to \tau\tau$ events, the Higgs Boson Machine Learning Challenge. The training data for this event contains 250,000 labeled, simulated ATLAS

events in `csv` format described here and here. You can download it yourself, but we will only play with a small subset (10k events).

2.0.4 Data handling

We'll use Pandas.

2.0.5 Installing XGBoost

Assuming you have Python, NumPy, Matplotlib, and Pandas installed, you may need to install XGBoost if it's not already installed.

```
pip install xgboost --user
```

2.0.6 Links

A lot of this was borrowed from other sources. These sources and other good places for information about XGBoost and BDTs in general are here: * XGBoost demo: Example of how to use XGBoost Python module to run Kaggle Higgs competition * Blog post by phunther: Winning solution of Kaggle Higgs competition: what a single model can do? * XGBoost Kaggle Higgs solution: https://github.com/hetong007/higgsml

2.0.7 Note

You will not be asked to do the model training yourselves. The train/test datasets will be processed and provided, with the XGBoost model trained ready to use. The main purpose of this problem is to get real hands-on practice of evaluating a model, like ROC, recall, etc.

2.1 XGBoost Tutorial

[ ]:
```
%pip install xgboost
```

[8]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb

%matplotlib inline
```

2.1.1 Data

Load data First, load in the data and look at it. We will download a 10k event subsample of the Kaggle training data. Then we'll put it in the right format for XGBoost.

[ ]:
```
! wget https://raw.githubusercontent.com/k-woodruff/bdt-tutorial/master/data/
↪training_10k.csv -O data/training_10k.csv
```

[9]:
```
data = pd.read_csv("training_10k.csv")
```

Let's see what the data looks like:

[10]:
```
print("Size of data: {}".format(data.shape))
print("Number of events: {}".format(data.shape[0]))
print("Number of columns: {}".format(data.shape[1]))

print("\nList of features in dataset:")
for col in data.columns:
    print(col)
```

```
Size of data: (10000, 33)
Number of events: 10000
Number of columns: 33

List of features in dataset:
EventId
DER_mass_MMC
DER_mass_transverse_met_lep
DER_mass_vis
DER_pt_h
DER_deltaeta_jet_jet
DER_mass_jet_jet
DER_prodeta_jet_jet
DER_deltar_tau_lep
DER_pt_tot
DER_sum_pt
DER_pt_ratio_lep_tau
DER_met_phi_centrality
DER_lep_eta_centrality
PRI_tau_pt
PRI_tau_eta
PRI_tau_phi
PRI_lep_pt
PRI_lep_eta
PRI_lep_phi
PRI_met
PRI_met_phi
PRI_met_sumet
PRI_jet_num
PRI_jet_leading_pt
PRI_jet_leading_eta
PRI_jet_leading_phi
PRI_jet_subleading_pt
PRI_jet_subleading_eta
PRI_jet_subleading_phi
PRI_jet_all_pt
Weight
Label
```

2.1.2 Detailed description of features

Prefix-less variables `EventId`, `Weight`, and `Label` have a special role and should not be used as input to the classifier. The variables prefixed with PRI (for PRImitives) are "raw" quantities about the bunch collision as measured by the detector, essentially the momenta of particles. Variables prefixed with DER (for DERived) are quantities computed from the primitive features. These quantities were selected by the physicists of ATLAS in the reference document either to select regions of interest or as features for the Boosted Decision Trees used in this analysis. In addition: * Variables are floating point unless specified otherwise. * All azimuthal ϕ angles are in radian in the $[-\pi, +\pi]$ range. * Energy, mass, momentum are all in GeV * All other variables are unitless. * Variables are indicated as "may be undefined" when it can happen that they are meaningless or cannot be computed; in this case, their value is -999.0, which is outside the normal range of all variables. * The mass of particles has not been provided, as it can safely be neglected for the Challenge.

Features: - `EventId`: An unique integer identifier of the event. Not to be used as a feature. - `DER_mass_MMC`: The estimated mass m_H of the Higgs boson candidate, obtained through a probabilistic phase space integration (may be undefined if the topology of the event is too far from the expected topology) - `DER_mass_transverse_met_lep`: The transverse mass (21) between the missing transverse energy and the lepton. - `DER_mass_vis`: The invariant mass (20) of the hadronic tau and the lepton. - `DER_pt_h`: The modulus (19) of the vector sum of the transverse momentum of the hadronic tau, the lepton, and the missing transverse energy vector. - `DER_deltaeta_jet_jet`: The absolute value of the pseudorapidity separation (22) between the two jets (undefined if `PRI_jet_num` ≤ 1). - `DER_mass_jet_jet`: The invariant mass (20) of the two jets (undefined if `PRI_jet_num` ≤ 1). - `DER_prodeta_jet_jet`: The product of the pseudorapidities of the two jets (undefined if `PRI_jet_num` ≤ 1). - `DER_deltar_tau_lep`: The R separation (23) between the hadronic tau and the lepton. - `DER_pt_tot`: The modulus (19) of the vector sum of the missing transverse momenta and the transverse momenta of the hadronic tau, the lepton, the leading jet (if `PRI_jet_num` ≥ 1) and the subleading jet (if `PRI_jet_num` $= 2$) (but not of any additional jets). - `DER_sum_pt`: The sum of the moduli (19) of the transverse momenta of the hadronic tau, the lepton, the leading jet (if `PRI_jet_num` ≥ 1) and the subleading jet (if `PRI_jet_num` $= 2$) and the other jets (if `PRI_jet_num` $= 3$). - `DER_pt_ratio_lep_tau`: The ratio of the transverse momenta of the lepton and the hadronic tau. - `DER_met_phi_centrality`: The centrality of the azimuthal angle of the missing transverse energy vector w.r.t. the hadronic tau and the lepton $C = \frac{A+B}{A^2+B^2}$ where $A = \sin(\phi_{\text{met}} - \phi_{\text{lep}})$, $B = \sin(\phi_{\text{had}} - \phi_{\text{met}})$, and ϕ_{met}, ϕ_{lep}, and ϕ_{had} are the azimuthal angles of the missing transverse energy vector, the lepton, and the hadronic tau, respectively. The centrality is $\sqrt{2}$ if the missing transverse energy vector $\vec{E}_{\text{T}}^{\text{miss}}$ is on the bisector of the transverse momenta of the lepton and the hadronic tau. It decreases to 1 if $\vec{E}_{\text{T}}^{\text{miss}}$ is collinear with one of these vectors and it decreases further to $-\sqrt{2}$ when $\vec{E}_{\text{T}}^{\text{miss}}$ is exactly opposite to the bisector. - `DER_lep_eta_centrality`: The centrality of the pseudorapidity of the lepton w.r.t. the two jets (undefined if `PRI_jet_num` ≤ 1) $\exp\left[\frac{-4}{(\eta_1-\eta_2)^2}\left(\eta_{\text{lep}} - \frac{\eta_1+\eta_2}{2}\right)^2\right]$ where η_{lep} is the pseudorapidity of the lepton and η_1 and η_2 are the pseudorapidities of the two jets. The centrality is 1 when the lepton is on the bisector of the two jets, decreases to $1/e$ when it is collinear to one of the jets, and decreases further to zero at infinity. - `PRI_tau_pt`: The transverse momentum $\sqrt{p_x^2 + p_y^2}$ of the hadronic tau. - `PRI_tau_eta`: The pseudorapidity η of the hadronic tau. - `PRI_tau_phi`: The azimuth angle ϕ of the hadronic tau. - `PRI_lep_pt`: The transverse momentum $\sqrt{p_x^2 + p_y^2}$ of the lepton (electron or muon). - `PRI_lep_eta`: The pseudorapidity η of the lepton. -

`PRI_ep_phi`: The azimuth angle ϕ of the lepton. - `PRI_met`: The missing transverse energy $E_{\mathrm{T}}^{\mathrm{miss}}$ - `PRI_met_phi` The azimuth angle ϕ of the missing transverse energy. - `PRI_met_sumet`: The total transverse energy in the detector. - `PRI_jet_num`: The number of jets (integer with value of 0, 1, 2 or 3; possible larger values have been capped at 3). - `PRI_jet_leading_pt`: The transverse momentum $\sqrt{p_x^2 + p_y^2}$ of the leading jet, that is the jet with largest transverse momentum (undefined if `PRI_jet_num` $= 0$). - `PRI_jet_leading_eta`: The pseudorapidity η of the leading jet (undefined if `PRI_jet_num` $= 0$). - `PRI_jet_leading_phi`: The azimuth angle ϕ of the leading jet (undefined if `PRI_jet_num` $= 0$). - `PRI_jet_subleading_pt`: The transverse momentum $\sqrt{p_x^2 + p_y^2}$ of the leading jet, that is, the jet with second largest transverse momentum (undefined if `PRI_jet_num` ≤ 1). - `PRI_jet_subleading_eta`: The pseudorapidity η of the subleading jet (undefined if `PRI_jet_num` ≤ 1). - `PRI_jet_subleading_phi`: The azimuth angle ϕ of the subleading jet (undefined if `PRI_jet_num` ≤ 1). - `PRI_jet_all_pt`: The scalar sum of the transverse momentum of all the jets of the events. - `Weight`: The event weight w_i, explained in Section 3.3. Not to be used as a feature. Not available in the test sample. - `Label`: The event label (string) $y_i \in \{s, b\}$ (s for signal, b for background). Not to be used as a feature. Not available in the test sample.

The data set has 10,000 events with 33 columns each. The first column is an identifier, and should not be used as a feature. The last two columns `Weight` and `Label`, are the weights and labels from the simulation, and also should not be used as features (this information is all contained in the documentation).

Now we can look at how many events are signal and background:

[11]:
```
# look at column labels --- notice last one is "Label" and first is "EventId"␣
  ↪also "Weight"
print(f"Number of signal events: {len(data[data.Label == 's'])}")
print(f"Number of background events: {len(data[data.Label == 'b'])}")
print(f"Fraction signal: {len(data[data.Label == 's'])/(len(data[data.Label ==␣
  ↪'s']) + len(data[data.Label == 'b']))}")
```

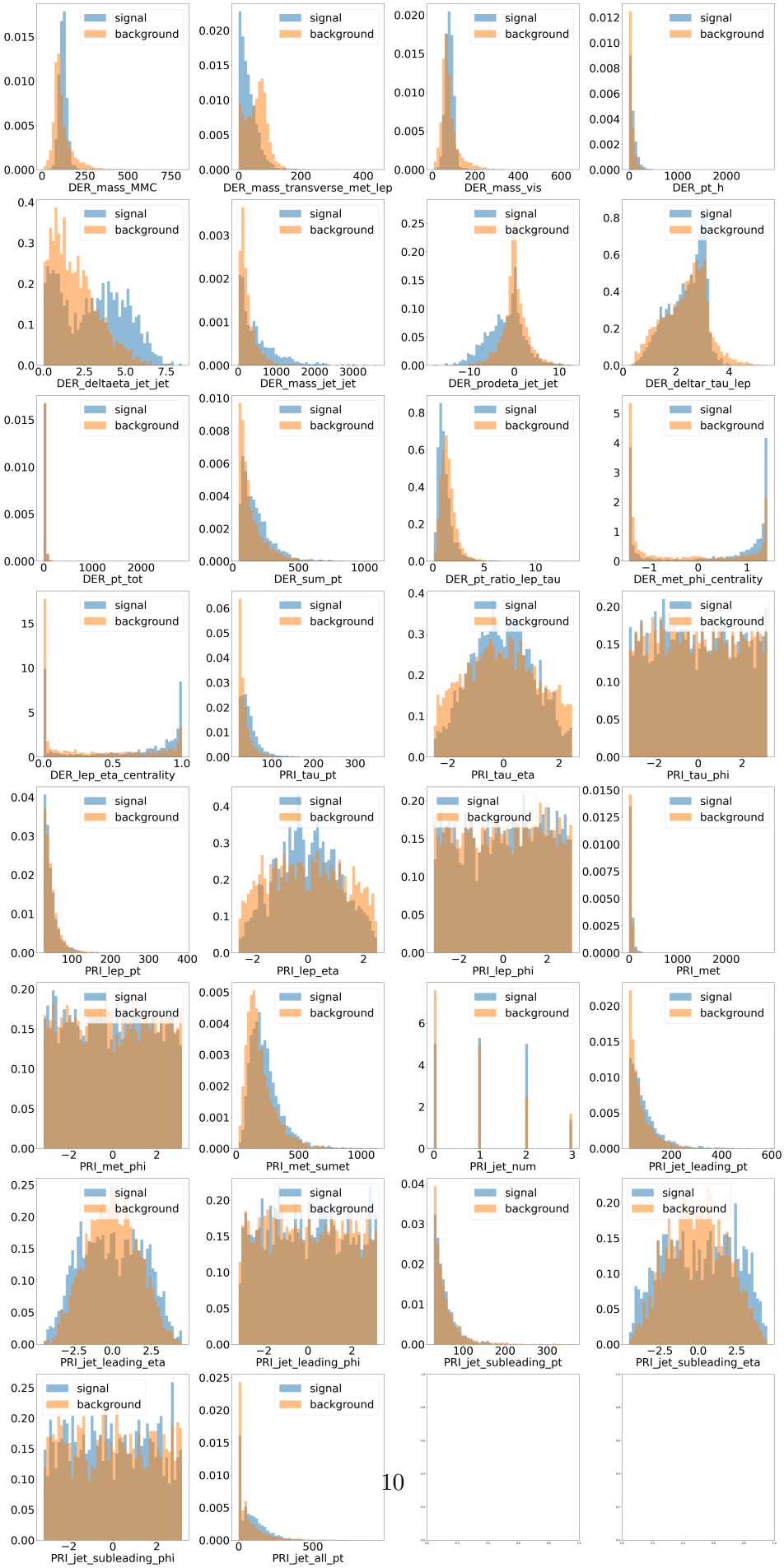
```
Number of signal events: 3372
Number of background events: 6628
Fraction signal: 0.3372
```

Visualize the features:

[12]:
```
plt.figure()


fig, axs = plt.subplots(8, 4, figsize=(40, 80))

for ix, ax in enumerate(axs.reshape(-1)):
    col = data.columns[ix + 1]
    if col == "Weight" or col == "Label":
        continue
    signal = data[col][data.Label == "s"].to_numpy()
    mask_signal = signal > -999
    background = data[col][data.Label == "b"].to_numpy()
```

```
    mask_background = background > -999
    xmin = min(np.min(background[mask_background]), np.min(signal[mask_signal]))
    xmax = max(np.max(background[mask_background]), np.max(signal[mask_signal]))

    ax.hist(signal[mask_signal], bins=np.linspace(xmin, xmax, 51), alpha=0.5,␣
 ↪label="signal", density=True)
    ax.hist(background[mask_background], bins=np.linspace(xmin, xmax, 51),␣
 ↪alpha=0.5, label="background", density=True)

    ax.set_xlabel(col, fontsize=40)
    ax.set_xlabel(col, fontsize=40)
    ax.tick_params(axis="both", which="major", labelsize=40)
    ax.legend(fontsize=40)
plt.tight_layout()
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```


signal
background
DER_mass_MMC
DER_mass_transverse_met_lep
DER_mass_vis
DER_pt_h
DER_deltaeta_jet_jet
DER_mass_jet_jet
DER_prodeta_jet_jet
DER_deltar_tau_lep
DER_pt_tot
DER_sum_pt
DER_pt_ratio_lep_tau
DER_met_phi_centrality
DER_lep_eta_centrality
PRI_tau_pt
PRI_tau_eta
PRI_tau_phi
PRI_lep_pt
PRI_lep_eta
PRI_lep_phi
PRI_met
PRI_met_phi
PRI_met_sumet
PRI_jet_num
PRI_jet_leading_pt
PRI_jet_leading_eta
PRI_jet_leading_phi
PRI_jet_subleading_pt
PRI_jet_subleading_eta
PRI_jet_subleading_phi
PRI_jet_all_pt

Format data: Now we should get the data into an XGBoost-friendly format. We can create DMatrix objects that will be used to train the BDT model. For now, we'll use all 30 of the features for training.

First, we'll slice up the data into training and testing sets. Here, we take 20% for the test set, which is arbitrary.

In this file, all samples are independent and ordered randomly, so we can just grab a chunk. Check out Scikit-learn Cross-validation for dividing up samples responsibly.

We can also change the data type of the `Label` column to the Pandas type `category` for easier use later.

[13]:
```
data["Label"] = data.Label.astype("category")
```

[14]:
```
data_train = data[:8000]
data_test = data[8000:]
```

Check to make sure we did it right:

[15]:
```
print(f"Number of training samples: {len(data_train)}")
print(f"Number of testing samples: {len(data_test)}")
print()
print(f"Number of signal events in training set: {len(data_train[data_train.
 ↪Label == 's'])}")
print(f"Number of background events in training set: {len(data_train[data_train.
 ↪Label == 'b'])}")
print(
    f"Fraction signal: {len(data_train[data_train.Label == 's'])/
 ↪(len(data_train[data_train.Label == 's']) + len(data_train[data_train.Label␣
 ↪== 'b']))}"
)
```

```
Number of training samples: 8000
Number of testing samples: 2000

Number of signal events in training set: 2688
Number of background events in training set: 5312
Fraction signal: 0.336
```

The `DMatrix` object takes as arguments: - `data`: the features - `label`: `1/0` or `True/False` for binary data (we have to convert our label to boolean from string `"s"/"b"`) - `missing`: how missing values are represented (here as `-999.0`) - `feature_names`: the names of all the features (optional)

[16]:
```
feature_names = list(data.columns[1:-2])  # we skip the first and last two␣
 ↪columns because they are the ID, weight, and label
```

```
print(len(feature_names))

train = xgb.DMatrix(
    data=data_train[feature_names], label=data_train.Label.cat.codes,␣
 ↪missing=-999.0, feature_names=feature_names
)
test = xgb.DMatrix(
    data=data_test[feature_names], label=data_test.Label.cat.codes,␣
 ↪missing=-999.0, feature_names=feature_names
)
```

```
30
```

Check if we did it right:

[17]:
```
print(f"Number of training samples: {train.num_row()}")
print(f"Number of testing samples: {test.num_row()}")
print()
print(f"Number of signal events in training set: {len(np.where(train.
 ↪get_label())[0])}")
```

```
Number of training samples: 8000
Number of testing samples: 2000

Number of signal events in training set: 2688
```

2.1.3 Make the model

Set hyperparameters: The XGBoost hyperparameters are defined here. For a nice description of what they all mean, and tips on tuning them, see this guide.

In general, the tunable parameters in XGBoost are the ones you would see in other gradient boosting libraries. Here, they fall into three categories: 1. General parameters: e.g., which booster to use, number of threads. We won't mess with these here. 2. Booster parameters: Tune the actual boosting, e.g., learning rate. These are the ones to optimize. 3. Learning task parameters: Define the objective function and the evaluation metrics.

Here, we will use the defaults for most parameters and just set a few to see how it's done. The parameters are passed in as a dictionary or list of pairs.

Make the parameter dictionary:

[18]:
```
param = {}

param["seed"] = 42  # set seed for reproducibility

# Booster parameters
param["eta"] = 0.1  # learning rate
param["max_depth"] = 10  # maximum depth of a tree
param["subsample"] = 0.8  # fraction of events to train tree on
```

```
param["colsample_bytree"] = 0.8  # fraction of features to train tree on

# Learning task parameters
param["objective"] = "binary:logistic"  # objective function
param["eval_metric"] = "error"  # evaluation metric for cross validation, note:␣
 ↪last one is used for early stopping
param = list(param.items())

num_trees = 100  # number of trees to make
```

First, we set the booster parameters. Again, we just chose a few here to experiment with. These are the parameters to tune to optimize your model. Generally, there is a trade off between speed and accuracy. 1. `eta` is the learning rate. It determines how much to change the data weights after each boosting iteration. The default is 0.3. 2. `max_depth` is the maximum depth of any tree. The default is 6. 3. `subsample` is the fraction of events used to train each new tree. These events are randomly sampled each iteration from the whole sample set. The default is 1 (use every event for each tree). 4. `colsample_bytree` is the fraction of features available to train each new tree. These features are randomly sampled each iteration from the whole feature set. The default is 1.

Next, we set the learning objective to `binary:logistic`. So, we have two classes that we want to score from 0 to 1. The `eval_metric` parameters set what we want to monitor when doing cross validation. (We aren't doing cross validation in this example, but we should be!) If you want to watch more than one metric, `param` must be a list of pairs, instead of a dict. Otherwise, we would just keep resetting the same parameter.

Last, we set the number of trees to 100. Usually, you would set this number high, and choose a cut off point based on the cross validation. The number of trees is the same as the number of iterations.

2.1.4 Now train!

[19]:
```
booster = xgb.train(param, train, num_boost_round=num_trees)
```

We now have a trained model. The next step is to look at it's performance and try to improve the model if we need to. We can try to improve it by improving/adding features, adding more training data, using more boosting iterations, or tuning the hyperparameters (ideally in that order).

First, let's look at how it does on the test set:

[20]:
```
print(booster.eval(test))
```

```
[0]     eval-error:0.17699999999999999
```

Okay, now we get the trained model, but how does it perform on the test set? These are the evaluation metrics that we stored in the parameter set.

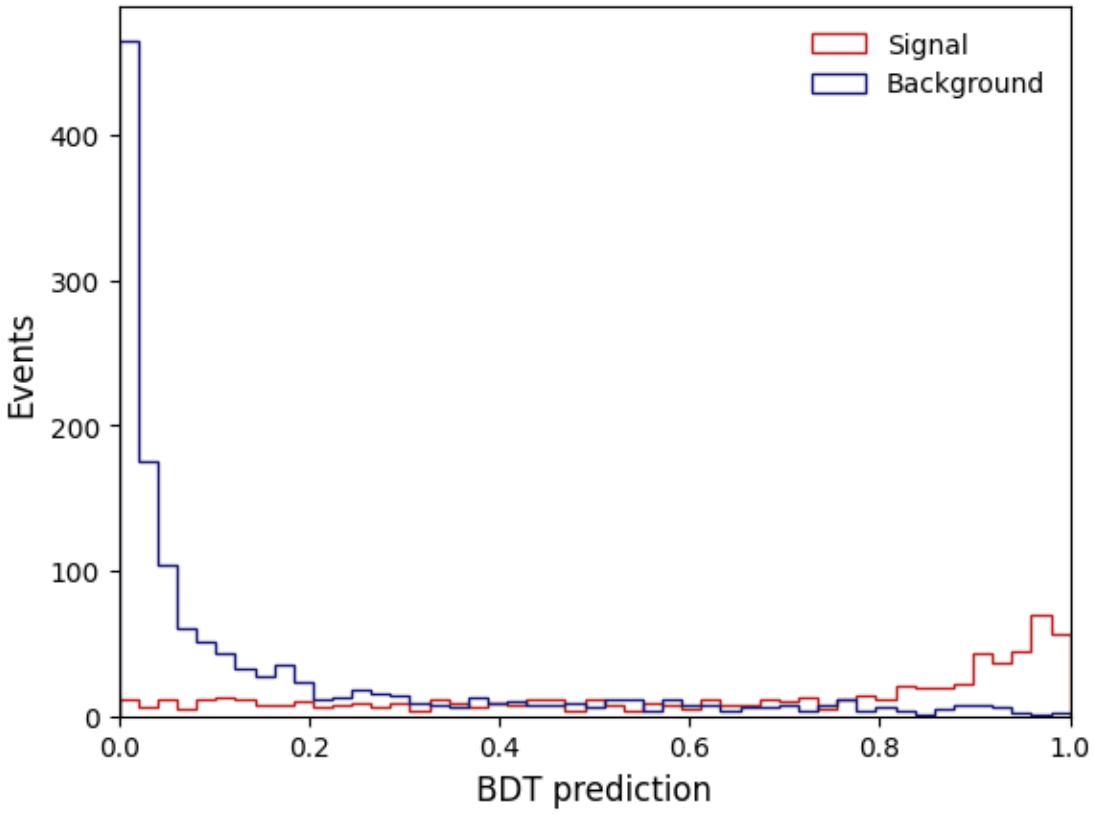
It's pretty hard to interpret the performance of a classifier from a few number. So, let's look at the predictions for the entire test set.

3 Part 1

Use the trained model (booster) to make predictions on the test set.

[43]:
```
# Implement your code here
predictions = booster.predict(test)
labels = test.get_label().astype(bool)
```

[44]:
```
# Plot signal and background predictions, separately
plt.figure()
plt.hist(predictions[labels], bins=np.linspace(0, 1, 50), histtype="step",␣
 ↪color="firebrick", label="Signal")
plt.hist(predictions[~labels], bins=np.linspace(0, 1, 50), histtype="step",␣
 ↪color="midnightblue", label="Background")
# Make the plot readable
plt.xlabel("BDT prediction", fontsize=12)
plt.ylabel("Events", fontsize=12)
plt.legend(frameon=False)
plt.xlim(0, 1)
plt.show()
```

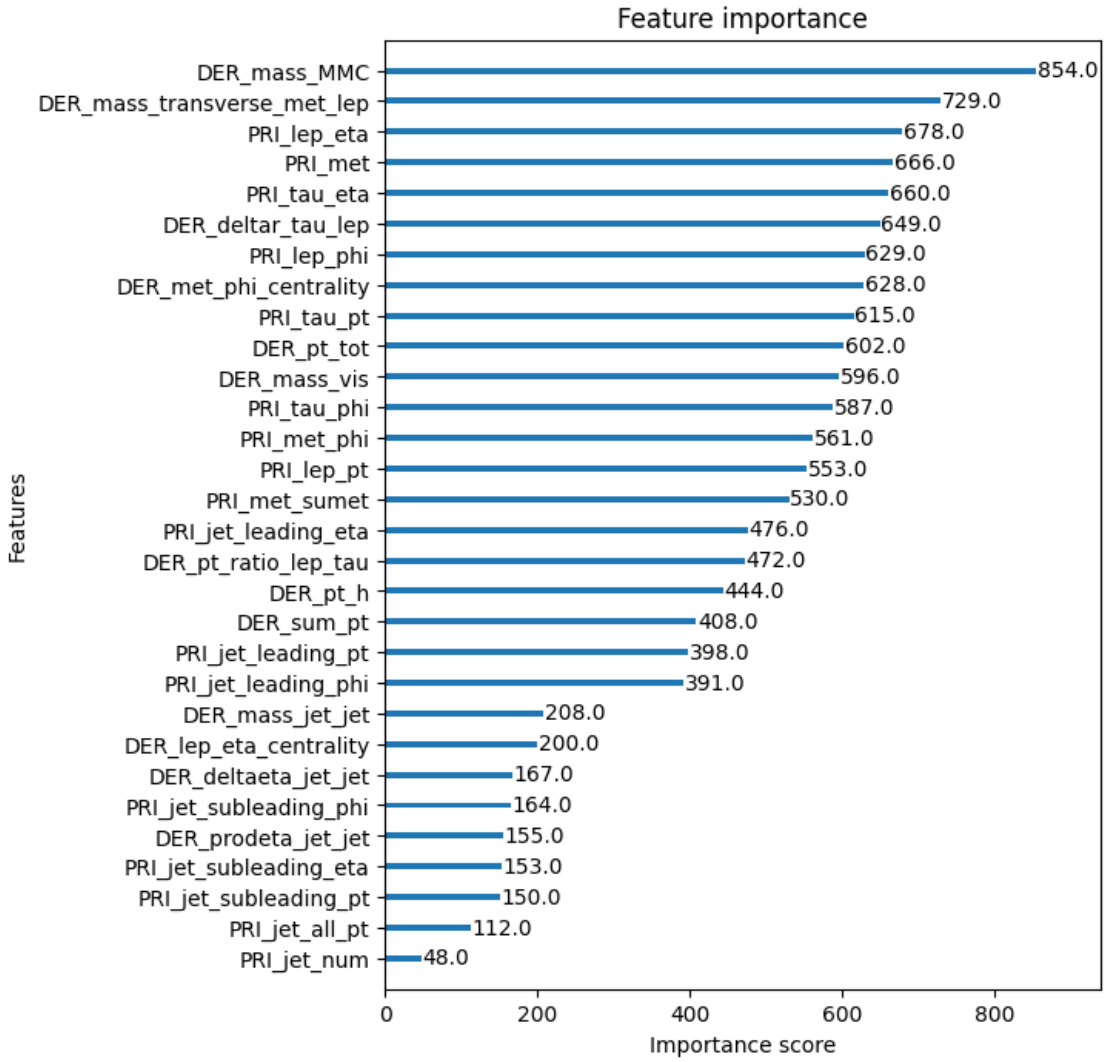


It's also very informative to look at the importance of each feature. The "F score" is the number

of times each feature is used to split the data over all the trees (times the weight of that tree).

There is a built-in function in the XGBoost Python API to easily plot this.

[32]:
```
fig, ax = plt.subplots(figsize=(6, 8))
xgb.plot_importance(booster, ax=ax, grid=False)
plt.show()
```

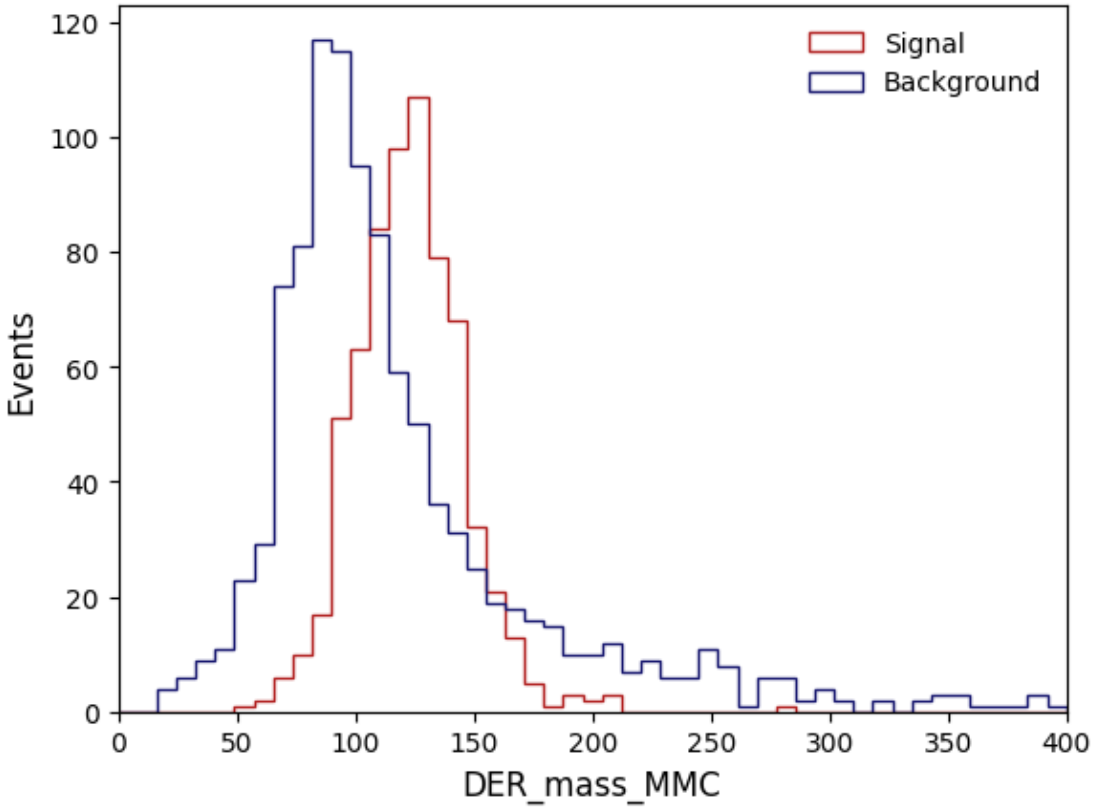


The feature that was used the most was `DER_mass_MMC`.

We can plot how this feature is distributed for the signal and background.

[33]:
```
plt.figure()
plt.hist(
    data_test.DER_mass_MMC[data_test.Label == "s"],
    bins=np.linspace(0, 400, 50),
```

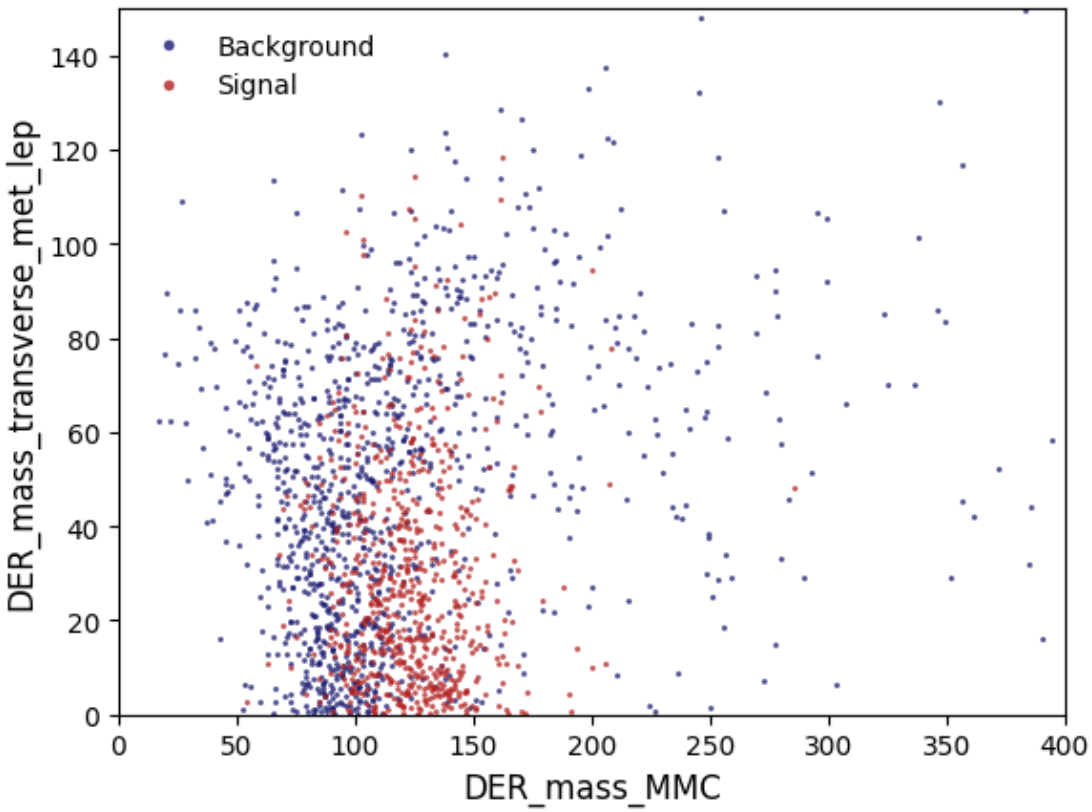
```
    histtype="step",
    color="firebrick",
    label="Signal",
)
plt.hist(
    data_test.DER_mass_MMC[data_test.Label == "b"],
    bins=np.linspace(0, 400, 50),
    histtype="step",
    color="midnightblue",
    label="Background",
)

plt.xlim(0, 400)
plt.xlabel("DER_mass_MMC", fontsize=12)
plt.ylabel("Events", fontsize=12)
plt.legend(frameon=False)
plt.show()
```



This variable is physically significant because it represents an estimate of the Higgs boson mass. For signal, it is expected to peak at 125 GeV. We can also plot it with one of the next most important features `DER_mass_transverse_met_lep`. Note: the exact ranking of features can depend on the

random seed and other hyperparameters.

[34]:
```
plt.figure()

mask_b = np.array(data_test.Label == "b")
mask_s = np.array(data_test.Label == "s")

DER_mass_MMC = np.array(data_test.DER_mass_MMC)
DER_mass_transverse_met_lep = np.array(data_test.DER_mass_transverse_met_lep)

plt.plot(
    DER_mass_MMC[mask_b],
    DER_mass_transverse_met_lep[mask_b],
    "o",
    markersize=2,
    color="midnightblue",
    markeredgewidth=0,
    alpha=0.8,
    label="Background",
)
plt.plot(
    DER_mass_MMC[mask_s],
    DER_mass_transverse_met_lep[mask_s],
    "o",
    markersize=2,
    color="firebrick",
    markeredgewidth=0,
    alpha=0.8,
    label="Signal",
)
plt.xlim(0, 400)
plt.ylim(0, 150)
plt.xlabel("DER_mass_MMC", fontsize=12)
plt.ylabel("DER_mass_transverse_met_lep", fontsize=12)
plt.legend(frameon=False, numpoints=1, markerscale=2)
plt.show()
```



3.1 Part 2

Calculate the precision, recall and F1-score as we learned from the lecture. Assuming using the default threshold 0.5.

1. Precision measures how many of the positive predictions made by the model are actually correct.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

2. Recall (also called sensitivity or true positive rate) measures how many of the actual positive cases were correctly identified by the model.

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

3. F1-score is the harmonic mean of precision and recall, providing a single metric to balance the trade-off between them. It ranges from 0 to 1, where 1 indicates perfect precision and recall.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

[47]:
```
# Implement your code below

from sklearn.metrics import precision_score, recall_score, f1_score
```

```
predicted_classes = predictions > 0.5
precision = precision_score(labels, predicted_classes)
recall = recall_score(labels, predicted_classes)
f1_score = f1_score(labels, predicted_classes)
```

[50]:
```
print(precision, recall, f1_score)
```

```
0.7652733118971061 0.695906432748538 0.7289433384379785
```

Changed following values for 1e-2 tolerance.

[53]:
```
# Check results
import math
assert math.isclose(precision, 0.77, rel_tol=1e-2), f"Precision not correct!"
assert math.isclose(recall, 0.69, rel_tol=1e-2), f"Recall not correct!"
assert math.isclose(f1_score, 0.73, rel_tol=1e-2), f"F1-Score not correct!"
```

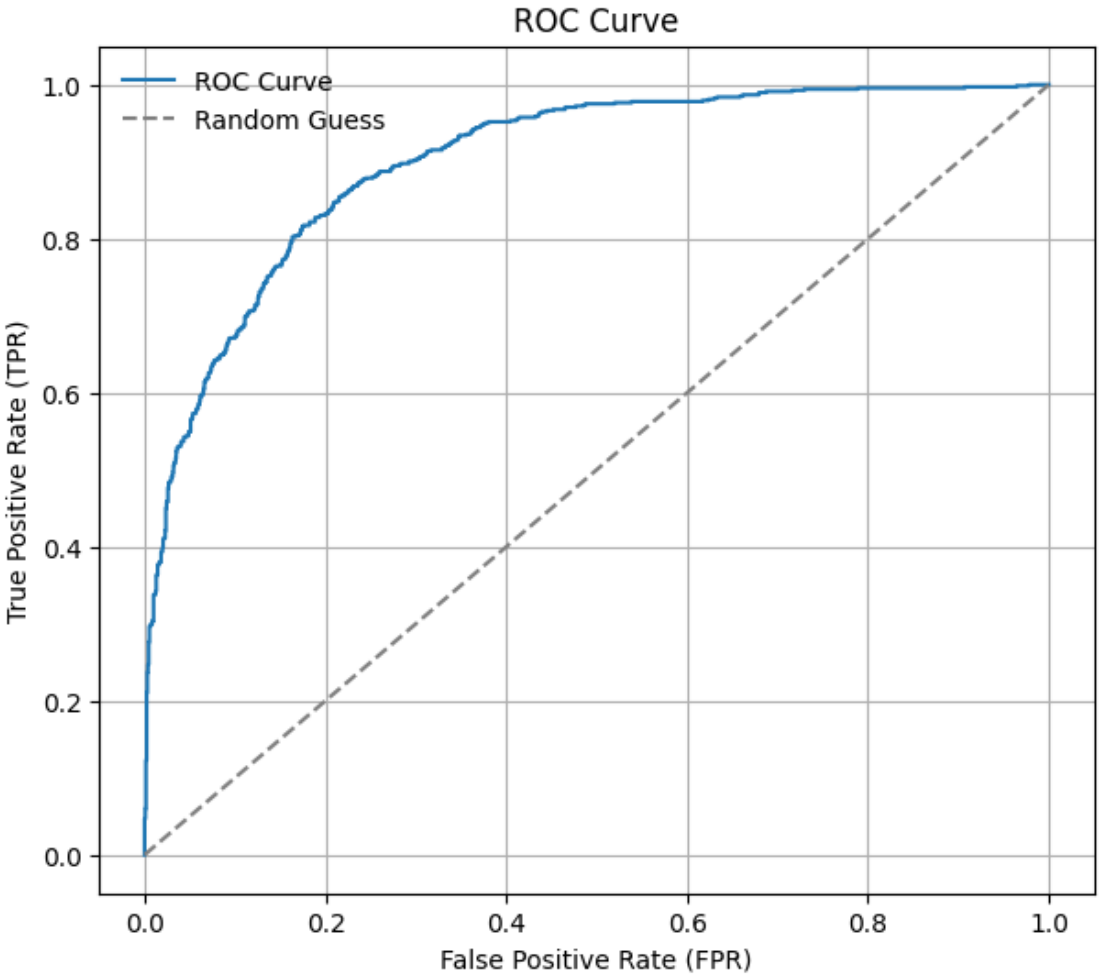
3.2 Part 3

Plot the ROC curve as we discussed during class with your own calculations, do not use the roc_curve library

[54]:
```
import numpy as np
import matplotlib.pyplot as plt

def compute_roc_curve(predictions, labels):
    # Input: predictions, labels
    # Output: fpr, tpr: arrays of FPRs and TPRs

    # Implement your code below
    sorted_indices = np.argsort(predictions)[::-1]
    sorted_predictions = predictions[sorted_indices]
    sorted_labels = labels[sorted_indices]

    thresholds = np.unique(sorted_predictions)
    thresholds = np.append(thresholds, thresholds[-1] + 1)
    thresholds = np.insert(thresholds, 0, -1)

    fpr = []
    tpr = []

    P = np.sum(labels == 1)
    N = np.sum(labels == 0)

    for threshold in thresholds:
        pred_pos = sorted_predictions >= threshold
```

```
        TP = np.sum(pred_pos & (sorted_labels == 1))
        tpr.append(TP / P if P > 0 else 0)

        FP = np.sum(pred_pos & (sorted_labels == 0))
        fpr.append(FP / N if N > 0 else 0)

    return np.array(fpr), np.array(tpr)
```

[55]:
```
fpr, tpr = compute_roc_curve(predictions, labels)

# Plot ROC curve
plt.figure(figsize=(7, 6))
plt.plot(fpr, tpr, linestyle='-', label="ROC Curve")
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label="Random Guess")
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR)")
plt.title("ROC Curve")
plt.legend(frameon=False)
plt.grid()
plt.show()
```


ROC Curve
ROC Curve
Random Guess
True Positive Rate (TPR)
False Positive Rate (FPR)
0.0
0.2
0.4
0.6
0.8
1.0