Summer Student Report

14 June - 3 September 2021

j.f.wack@sms.ed.ac.uk

Jet Clustering and Momentum Regression

Julian Wack

Supervisors: Clemens Lange and Jan Kieseler

Abstract

In high energy particle physics, jets are an important observable for theory testing as they connect the detected final state with the processes at the collision vertex. Due to their versatile employment, jets play a central role in most of the analyses conducted in HEP. Beginning in 2007 with the introduction of the k_t clustering algorithm, significant improvements in the performance and accuracy of jet clustering algorithms for pp collisions were achieved.
The aims of this project are two-fold. During the first part I investigated the anti-k_t clustering algorithm, representing the established jet reconstruction scheme used for pp collisions. Specifically, I reproduced the jet clustering done through CMS Software components (CMSSW) using the pyjet library, before attempting the same task using my own Python implementation of the anti-k_t algorithm. In the second part of the project, a novel jet momentum regression method is introduced, employing correction factors to the p_t of the jet constituents which are deduced by a Deep Neural Network. The comparison with an established regression method shows that with the currently trained model, the new method is not competitive.

Contents

1 Introduction

A typical high energy collision, like the proton-proton collisions at the LHC, produces energetic particles leaving the primary interaction vertex. It is convention to describe these processes in the parton model, where hadrons are assumed to contain point-like particles named 'partons'. In the quark model partons are matched to quarks and gluons. QCD effects allow energetic partons to stimulate pair-productions and emit gluons, both of which tend to move collinearly to the initial parton[1]. If these secondary particles are energetic enough, they repeat the same process, sparking a cascade of pair productions, decays, and gluon emissions. Such cascade is called a parton shower. As the spatial extent of the parton shower increases, α_S grows to order ~ 1, leading into the non-perturbative regime of QCD. Here the initially colour-carrying partons hadronize to colourless hadrons which will be eventually measured by the detector. The detected collection of collimated hadrons and fermions form the constituents of a jet. A jet can be thus be seen as a proxy for the initial parton as it groups together the terminal particles of the parton decay cascade. Additionally, jets also reduce the number of high level objects in an event, thereby simplifying theoretical computations ([1], [2]). Consequently, jets form the bridge between the predictions of QCD at the microscopic scale and measurements on the macroscopic level. Performing precision measurements, testing theory predictions, or discovering new physics therefore heavily relies on the ability to form correct jets. In this context 'correct' means to only group particles together that originated from the initial parton in question and to obtain a jet whose physical properties (namely its momentum and energy) are similar to the ones of the initial parton. Contamination of the signals from the primary partons through background particles (hereafter referred to as pileup) makes the task of clustering correct jets significantly more difficult. Sophisticated techniques, including Pileup Per Particle Identification (PUPPI) and Charged Hadron Subtraction (CHS), have been developed to largely resolve this issue.
The rules by which particles are grouped into jets are summarized in the jet definition and implemented in the jet algorithm. The two major classes of jet algorithms are cone and sequential recombination algorithms. The latter kind has been very popular in recent years due to its conceptual simplicity and theoretical properties. Namely, the jets clustered by a sequential recombination algorithm remain approximately the same when adding soft radiation to the event or when the trajectory of a particle is split into two collinear ones. When an algorithm satisfies these properties, it is regarded as infrared and collinear (IRC) safe. The jets produced by such algorithm are invariant under QCD effects and thus allow perturbative computations.

2 Jet clustering using the pyjet library

To gain some familiarity with the process of clustering jets and different forms of pileup mitigation, the beginning of the project was dedicated to reproducing the jet clustering obtained by CMSSW. Henceforth, the jets from this clustering will be referred to as 'recojets' (short for reconstructed jets). The results from CMSSW are written in a ROOT ntuple which will form the basis for the following analysis.
The pyjet library (see [3]) provides a Python interface for the FastJet package which contains a large variety of efficient tools for jet finding and is documented in [4]. Pyjet supplies ready-to-use clustering methods, allowing to quickly compare different jet clusterings. This enabled me to reconstruct the recojets and to verify that they have been obtained through the anti-k_t algorithm, have a minimum

[1]Please consult [1], pp. 8-9 for a quantitative explanation of this collinear motion.

p_t of 15 GeV, and have CHS subtraction applied. The jets clustered through the pyjet library will be referred to as 'pyjets' and have been built using the above jet algorithm specifications[2].

2.1 Matching algorithm

Before turning to the differences between recojets and CHS pyjets note that both sets of jets are returned sorted by descending p_t. Thus, the sequence in which the jets are presented might be influenced when slightly different particles are clustered. Hence it is important not to naively compare the $i-th$ recojet with the $i-th$ pyjet, but to employ a simple matching algorithm.
This function accepts 4 lists containing the values of η, ϕ for the two sets of jets we want to pair up. The metric used to determine whether two jets correspond to each other is their axis separation in the η, ϕ plane, given by $\Delta R = \sqrt{(\eta_1 - \eta_2)^2 + (\phi_1 - \phi_2)^2}$. Jets clustered from the pyjet method will be denoted with the subscript 1 while the values of the recojets will carry the subscript 2.
The matching algorithm begins by considering the hardest pyjet, finding ΔR to all recojets, and selecting the recojet which results in the smallest ΔR. Then the next hardest pyjet is considered until all pyjets have been matched to a recojet. Should the recojet leading to the smallest ΔR be already used in a pair with a harder pyjet, then the current pyjet will be paired with the recojet which provides the next smallest ΔR. This assures that the most dominant pyjets are paired correctly. The matching function returns a list of integers such that the integer j at position i represents the pairing of the i-th pyjet with the j-th recojet.
When $j = -1$, no corresponding recojet could be found. This can occur when there are more pyjets than recojets or when the smallest available ΔR for some pyjet is larger than $R_{\text{cutoff}} = 0.4$. This cut-off value was deduced by examining the maximal distance between correct pairs of reco- and genjets[3] (see figure 1a) and by trying to maximize the accuracy of the algorithm. The latter causes a cut-off value of 0.4 to be selected rather than 0.16 as one would expect from figure 1a alone.

2.1.1 Accuracy and errors

Using the correct paring between recojets and genjets as well as the result from the matching algorithm allows to compute the accuracy and errors of the algorithm. Figure 1b shows the accuracy of the algorithm by plotting the number of correct matches divided by the number of total matches for individual events. On average, one can expect that 98.23% of the matches for a given event are correct. This does not significantly differ from the frequency of 98.11% at which any given pair of jets are correctly matched.

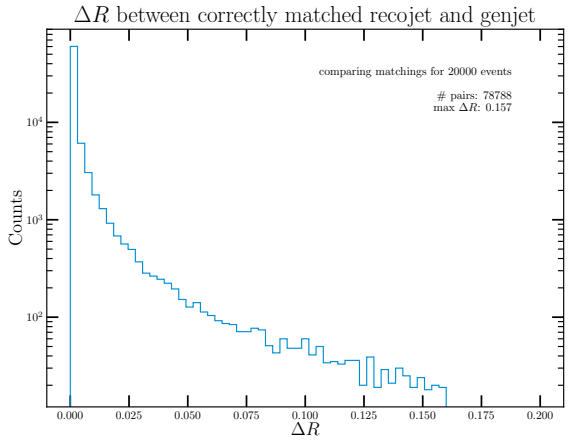
The algorithm can make 3 types of errors:

1. A recojet that does not correspond to any genjet (i.e. is a fake jet) gets mistakenly matched.
2. Declaring a recojet as fake even though it is not. Henceforth referred to as false-fake jets.
3. The wrong genjet is matched to a recojet.

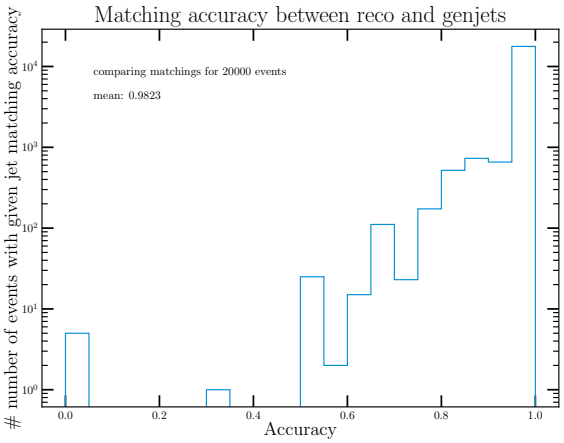
Figure 2 illustrates the significance of each error type for $R_{\text{cutoff}} = 0.4$. Error 3 has been omitted from figure 2 as no occurrences have been observed. Choosing $R_{\text{cutoff}} = 0.16$ doubles the frequency of error 2 while the rate at which error 1 occurs remains unchanged, consequently decreasing the algorithm's accuracy. In fact, through manual testing of 10 different R_{cutoff} values, I found that the accuracy saturates for $R_{\text{cutoff}} = 0.4$.

[2] A jet parameter of $R = 0.4$ was used in the anti-k_t algorithm.

[3] Genjets are the truly correct jets, deduced directly from the Monte Carlo simulation of the event. The correct pairing between recojets and genjets can be extracted from the ROOT ntuple.

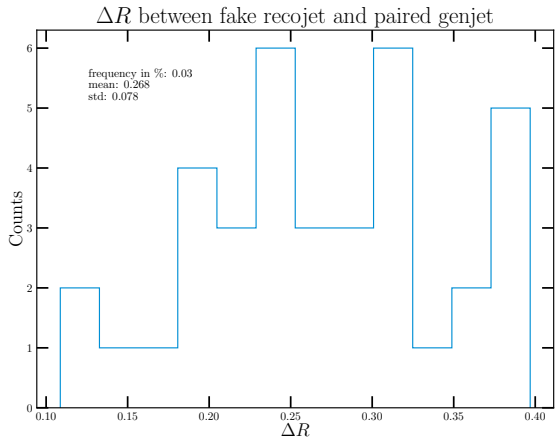


(a) ΔR cut-off

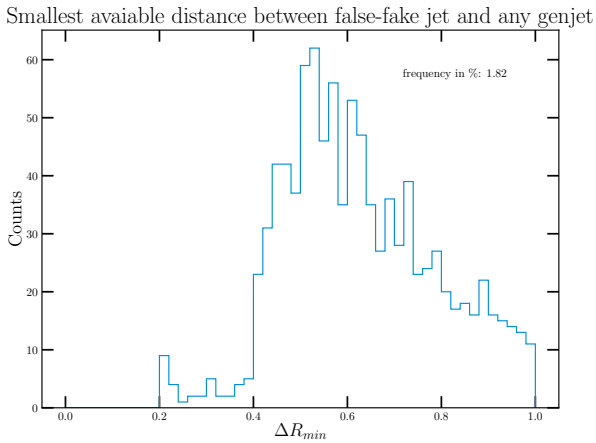


(b) Accuracy of matching algorithm

Figure 1: Deciding factors of choice of R_{cutoff}



(a) Error 1



(b) Error 2

Figure 2: Matching Algorithm errors

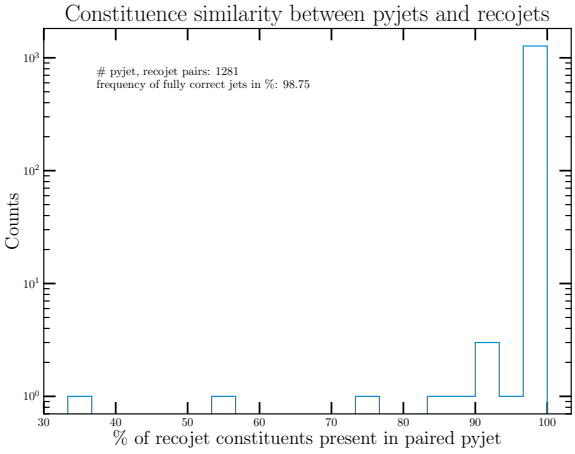


Figure 3: Similarity between recojets and pyjets

2.2 Clustering results

The results of the pyjet clustering are not adjusted for missing energy, while the recojets have Jet Energy Corrections (JECs) applied, meaning that the jet p_t and mass have been corrected by a factor. The correction factors can be extracted from the ntuple, allowing to deduce the pre-corrected recojet p_t and mass (later referred to as 'raw pt' and 'raw mass'). Only recojets whose corrected p_t is greater than 15 GeV are recorded. It is important to note that the correction factors are typically smaller than 1, causing a significant fraction of recojets with pre-corrected $p_t > 15\,\text{GeV}$ not to show in the ntuple. The pyjet clustering will contain all such jets, leading to a different number of recojets and pyjets. For 200 events, there are 1281 corrected recojets and 4042 pyjets with $p_t > 15\,\text{GeV}$. All these recojets have an associated pyjet[4]. Figure 3 shows that 98.75% of the pyjets contain the exact particles as their recojet partner.

Note that in figure 3, 'similarity' refers to the percentage of recojet constituents which are also included in the paired pyjet. Thus, if the considered pyjet contains all of the recojet particles and some additional ones, then this pair is regarded to have 100% similarity. Due to this flaw, the more meaningful statistic is the above quoted frequency that the pyjet-recojet pairs have identical constituents. Across all 1281 pyjet-recojet pairs, the two sets of jets differ in no more than 60 particles, which manifests itself in the similarity of the distributions of kinematic variables for jet constituents, as shown in figure 4. Consequently, the position and the momentum of the resulting jets are very similar as illustrated in figure 5, where the main source of differences are rounding errors.

3 Jet clustering using a NumPy based implementation of the anti-k_t algorithm

After gaining some familiarity with jet clustering on a high level, my attention shifted to studying the anti-k_t algorithm in detail by implementing the algorithm myself. Basing the algorithm on NumPy arrays makes it fully differentiable, which allows it to be directly integrated in a TensorFlow framework. Exploring this aspect further is a possible extension to this part of the project. Similar to

[4]Later referred to as pyjet-recojet pairs

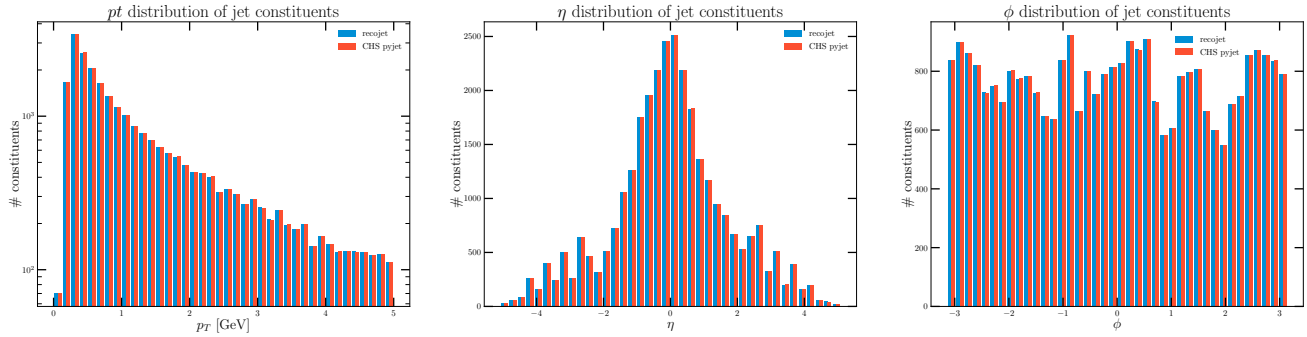


Figure 4: Kinematics of jet constituents

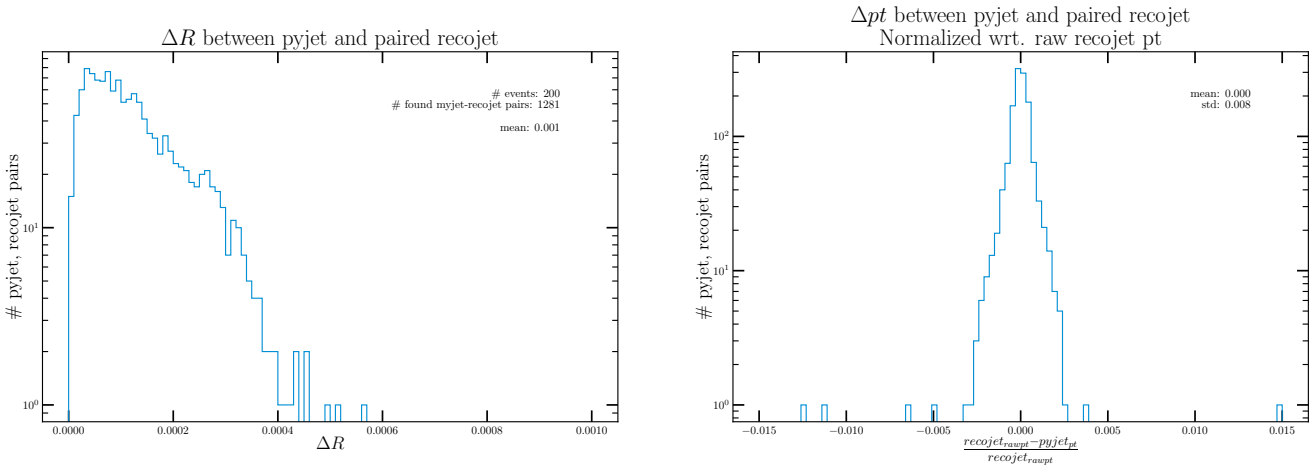


Figure 5: Difference in jet properties for pyjet-recojet pairs

the preceding section, the aim of this part of the project was to obtain the results from the CMSSW clustering. Due to the above-mentioned problem related to the energy correction of the recojets, I will compare the jets of the NumPy implementation (henceforth called 'myjets') to the pyjets. In this section particles will be described through their (p_t, η, ϕ, m) 4 Lorentz vectors which can be constructed from the ntuple.

3.1 The anti-k_t algorithm

Before discussing the comparison between myjets and pyjets, I give a high-level description of the operation principles of the anti-k_t algorithm. A more detailed discussion of the algorithm and its variations can be found in [4] and [5]. The anti-k_t algorithm belongs to the class of sequential recombination algorithms and has become the most commonly use jet algorithm at the LHC. The main characteristic of sequential jet algorithms is that pairs of particles are considered which are subsequently combined to new objects called pseudojets until an algorithm-specific termination condition is reached, after which the pseudojet is declared to be a jet. In the case of the anti-k_t algorithm, the combination and termination conditions are encoded by two distance-like measures d_{ij} and d_{iB}. The former describes the separation of the pair of pseudojets i and j, while the latter represents the separation of pseudojet i from the beam axis (conventionally chosen to be the z axis).
The anti-k_t algorithm operates as follows:

1. For every pair of pseudojets i, j, find the anti-k_t distance

$$d_{ij} = \min\left(\frac{1}{p_{ti}^2}, \frac{1}{p_{tj}^2}\right)\frac{\Delta R_{ij}^2}{R^2} \tag{1}$$

   where $\Delta R_{ij} = \sqrt{(y_i - y_j)^2 + (\phi_i - \phi_j)^2}$ is the Euclidean distance between i, j in the y, ϕ plane and y denotes the rapidity y of a particle. The rapidity can be determined via

$$y = \ln\left(\frac{\sqrt{m^2 + p_t^2\cosh\eta^2} + p_t\sinh\eta}{\sqrt{m^2 + p_t^2}}\right).$$

   The parameter R is commonly referred to as the jet-radius parameter as it roughly defines the maximal separation between the jet axis and any of its constituents. Usually, R is of order 1, with an conventional value of 0.4 which I will adapt.

2. For every pseudojet i, compute the beam distance:

$$d_{iB} = \frac{1}{p_{ti}^2} \tag{2}$$

3. Find the minimum of d_{ij} and d_{iB} (d_{ij}^{min} and d_{iB}^{min} respectively).
   Should $d_{ij}^{min} \leq d_{iB}^{min}$, combine i, j to a single pseudojet. There are various options to implement this combination mathematically, collectively known as recombination schemes. A brief overview of the most commonly used ones can be found in section 3.4 of [4]. The current standard in LHC experiments is the E-scheme through which the merged pseudojet is found by the unweighted 4-vector sum of i and j.
   Should $d_{iB}^{min} < d_{ij}^{min}$, then i is declared to be a jet and removed from the list of remaining pseudojets. It is common to introduce a p_t threshold at 15 GeV, such that jets with $p_t < 15$ GeV are disregarded.

4. Repeat the process until no pseudojets are left.

Observe that in this algorithm all initial particles will be clustered in some jet, leading to a large number jets where the majority are insignificant. The above mentioned p_t threshold assures that a more manageable and meaningful number of jets is considered.
The anti-k_t is a special case of a more general class of sequential jet algorithms, all following the logical described above, but with distance measures of the general form

$$d_{ij} = \min(p_{ti}^{2p}, p_{tj}^{2p})\frac{\Delta R_{ij}^2}{R^2} \quad \text{and} \quad d_{iB} = p_{ti}^{2p}, \quad p \in \mathbb{R}.$$

The three most common sequential jet algorithms are obtained through $p = 1, 0, -1$ and go by the names of 'k_t', 'Cambridge/Aachen', and 'anti-k_t' respectively. Choosing a negative value for p assures that hard particles get clustered before soft ones and produces conical jets when considered in isolation of hard background particles or other jets. A more detailed discussion of this class of algorithms can be found in [2].

3.2 Clustering results and problems

Overall, the correspondence between myjets and pyjets is excellent. The two sets differ in only 5 out of 4042 jets (by one particle in three cases and two particles in one case). These jets are visible as the outliers in figure 6a. Ignoring these outliers reveals the excellent correspondence between the myjets and the pyjets (see 6b). The magnitude of the the differences in 6b suggests that myjets are equivalent to pyjets up to rounding errors and the above mentioned outliers.

3.2.1 Degenerate minima

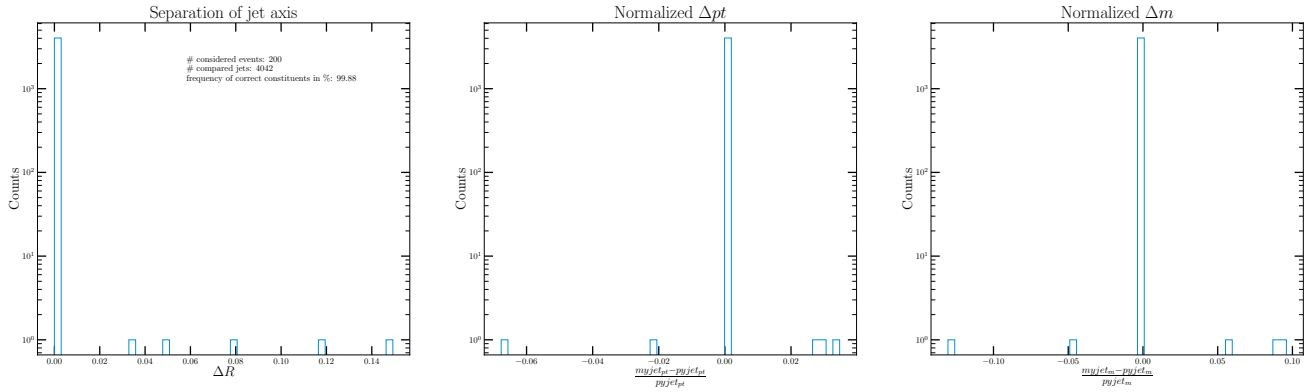
A problem that occurred in a significant portion of events but has not been addressed in the literature is that the minima of d_{ij}, d_{iB} can be degenerate. This is due to the limited precision at which variables are stored in the ntuple. For this project float-32 variables where used, giving a precision of 7 decimal places. I implemented the following logic to break the degeneracy by favouring hard and massive jets:

- Should d_{ij}^{min} be degenerate, choose the degenerate pair with the largest combined p_t.
- Should d_{iB}^{min} be degenerate, choose the degenerate pseudojet with the largest mass.

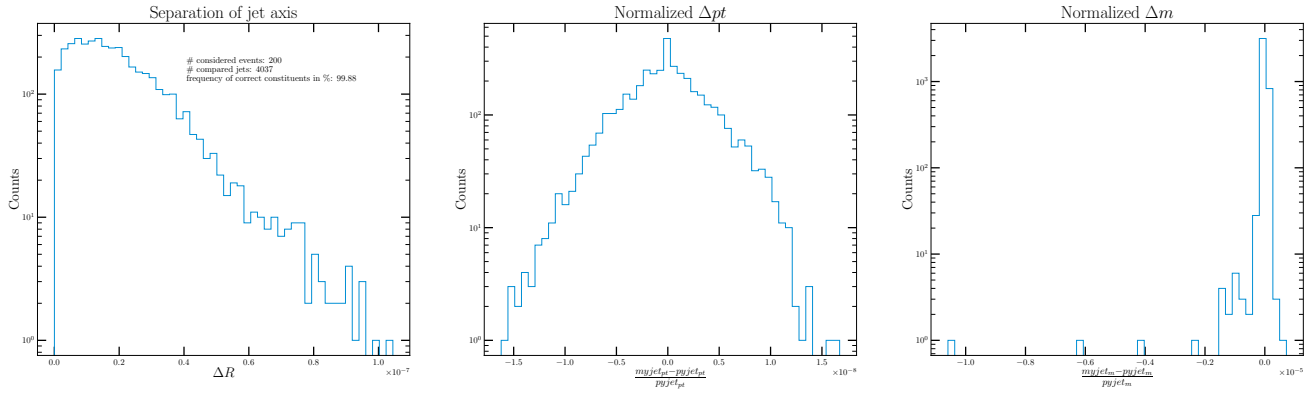
As I were unable to find how these degeneracies are treated in the FastJet library, a systematic difference is introduced between the clustering done by pyjet and the clustering through our NumPy implementation. Applying these decision rules corrected my clustering for events where it deviated significantly from the pyjet result, outweighing the systematic error introduced. This systematic error is the origin of the 5 outliers shown in 6a.

3.2.2 Precision of $4-$vector sum

An additional complication is rooted in the precision at which the sum of two (p_t, η, ϕ, m) Lorentz $4-$vectors is computed. Note that this $4-$vector is expressed in a mixed basis, meaning one cannot naively sum two such $4-$vectors by finding the sum of the components. The implementation of this sum is based on methods from the ROOT class 'TLorentzVector', documented in [6]. There exist various Python libraries like *awkward* that provide methods of summing these types of $4-$vectors,



(a) All differences



(b) Differences without the outliers

Figure 6: Differences between myjets and recojets

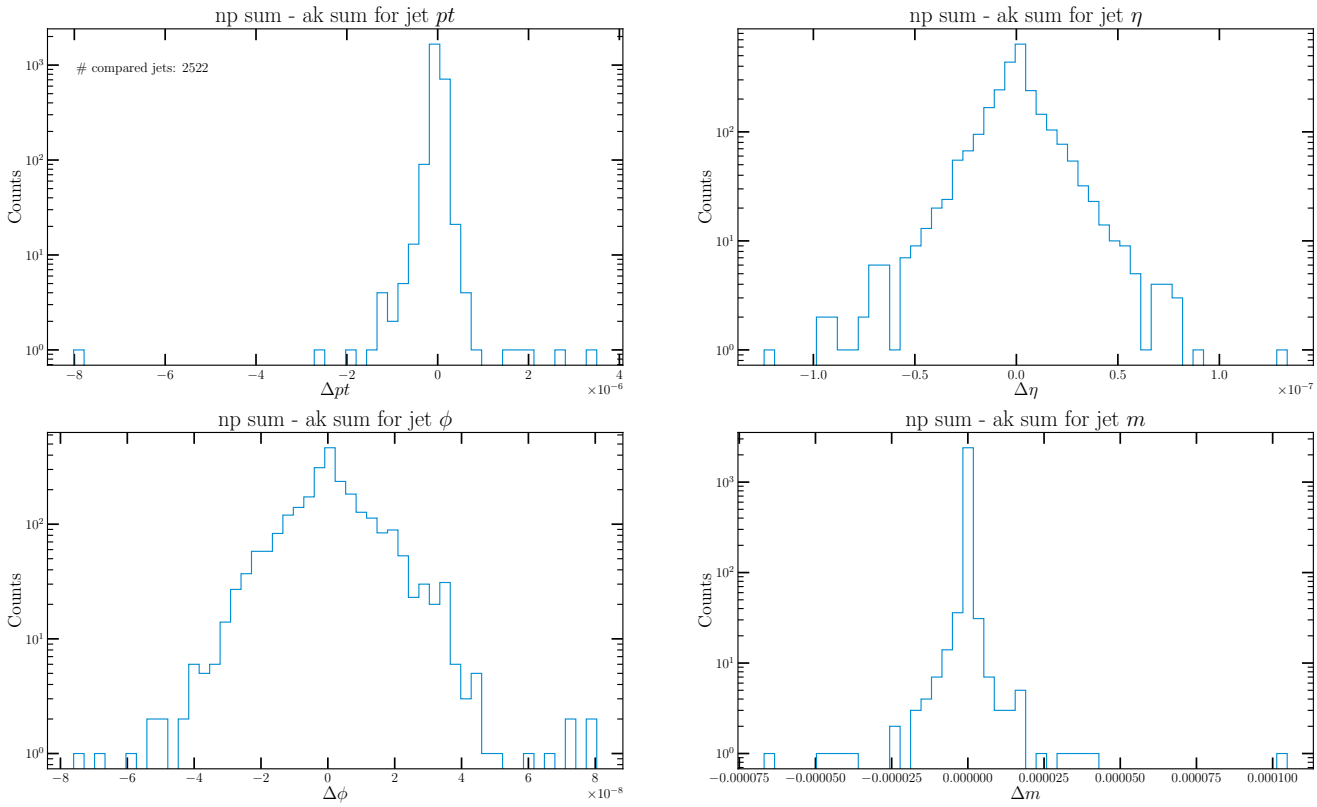


Figure 7: Accuracy of the NumPy (p_t, η, ϕ, m) 4-vector sum

which allowed to test the accuracy of the NumPy implementation. In the anti-k_t algorithm several 4−vector sums are computed in the clustering of a jet, meaning that the error in the summation algorithm amplifies over the course of the clustering. Consequently, it is vital to build a high precision 4−vector sum method. This was achieved through the use of the *mpmath* library (see [7]), computing the 4-vector sum at a precision of 100 decimal places without observing significant increase in the run time of the anti-k_t implementation. The results of comparing the NumPy 4-vector sum method (referred to as 'np sum') with the awkward method (referred to as 'ak sum') are shown in figure 7 and indicate that sufficient precision has been achieved.

3.3 Run time

The way the anti-k_t algorithm has been described above leads to a $\mathcal{O}(N^3)$ run time as finding all possible pseudojet pairs d_{ij}, and subsequently their minima, scales as $\mathcal{O}(N^2)$ which needs to be repeated N times until all particles have been clustered. The FastJet library uses complex optimisation methods (detailed in [5], section 2.3 under 'speed'), including a geometrical approach to the clustering task, to reduce the run time to $\mathcal{O}(N\log(N))$, provided $N \lesssim 20000$. For proton-proton collisions, the number of detected particles is well within this threshold. For larger events, as they occur in heavy ion collisions, the run time develops as $\mathcal{O}(N^{3/2})$.
Due to the limited time available to work on this project, I was unable to study these efficient

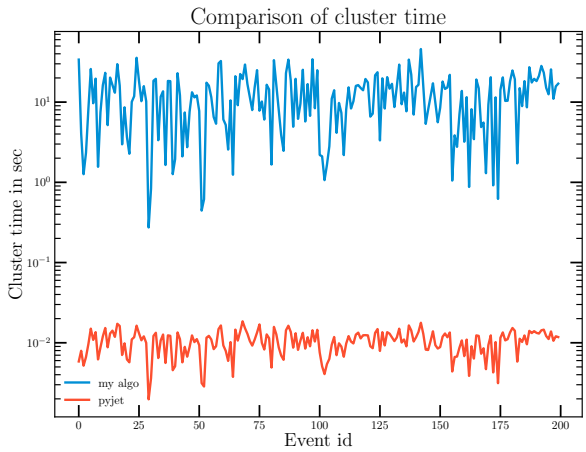


Figure 8: Cluster time of my anti-k_t implementation, compared to pyjet

methods to improve the speed of my clustering. Instead, I took three simple steps to decrease the run time per event of the NumPy implementation by a factor of roughly 20:

- Noticing that d_{ij} is a symmetric matrix with zeros on the diagonal means only $\frac{n(n-1)}{2}$ anti-k_t distances need to be calculated.
- Using Numba to significantly speed up explicit loops and specifically the computation of the anti-k_t distance d_{ij}.
- Rather than computing all d_{ij}, d_{iB} again when having passed through one iteration of the algorithm, I compute the set of distances for all initial particles once and then update only specific elements effected by the merging of two pseudojets or the removal of a pseudojet.

With these steps I obtain an average clustering time of roughly 13 seconds per event, compared to an average of 0.01 seconds attained by the pyjet implementation (averaged over 200 events). Apart from the far more sophisticated run time optimisations used in pyjet, the C++ code of FastJet is inherently faster than Python and NumPy code, resulting in a difference of 3 orders of magnitude. The cluster time for each considered event is displayed in figure 8.

4 Jet momentum regression using a DNN

After having studied a subsection of currently used jet clustering methods, in the final part of the project I investigate an alternative procedure to determine the jet p_t. The established recombination schemes all obtain the jet p_t through a weighted 4-vector sum of its constituents. As mentioned earlier, CMSSW uses the E-recombination scheme, corresponding to an unweighted sum. However, when comparing the recojet p_t with the genjet p_t, as it is done in figure 9, it becomes apparent that this form of jet momentum regression fails to give an accurate estimate of the true jet p_t in a considerable number of cases. This motivates the search for new momentum regression methods.
The method I considered deduces the jet p_t through a linear combination of the constituents' momenta. The coefficients of this sum are deduced through a Deep Neural Network (DNN), based on the *DeepJetCore* package, featuring two Graph network layers. The full code can be found in [8] and is derived from the example provided in DeepJetCore (see 'Usage' in [9]).

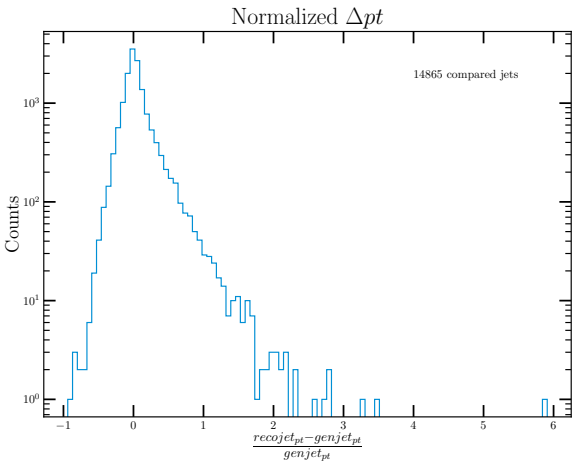


Figure 9: Normalized p_t difference between recojets and genjets

4.1 Results and suggestions

The network, whose architecture will be laid out below, has been trained on 63,000 jets with a learning rate of 10^{-5}. For the loss function, the square of the normalized difference between the predicted jet p_t and the genjet p_t is used. The development of the loss throughout the training epochs is shown in figure 10a. The stagnating validation loss indicates that the full predictive capacity of the model has been reached. Note that for testing, the model with the smallest validation loss has been chosen. The error of the network predictions for the test set[5] is shown in figure 10b. The peak at a normalized difference of -1 and the general dominance of negative Δp_t values, shows that the DNN jet momentum regression underestimates the jet p_t and typically predicts momenta which are just a fraction of the genjet p_t. The absence of a peak in the distribution around 0.0 encapsulates that the network was unable to learn relevant features for the regression task. Even though the peak in figure 9 is relatively wide, the conventional jet momentum regression provides the significantly higher predictive power out of the two compared regression methods.

There are various places where more work is needed to uncover the full potential of the DNN jet p_t regression. A non-exhaustive list includes:

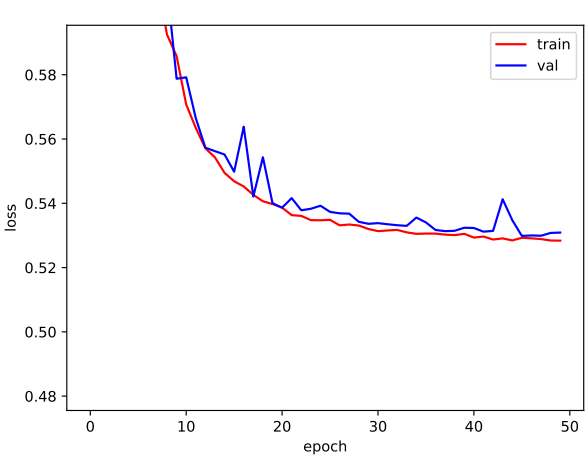
- The set of constituent features which are passed to the model.
- The zero padding which is used due to varying number of constituents in jets.
- The activation function of the final layer to produce greater weights and thus obtain a larger jet p_t.
- The network architecture.

4.2 Network architecture

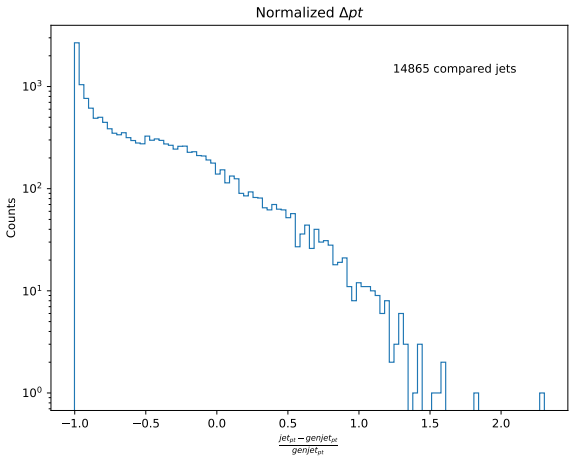
The DNN operates on individual jets[6] and takes a feature array containing characteristics of the jets' constituents as its input. The following particle properties are passed:

[5]The prediction is performed on the set of jets used in figure 9.

[6]During training and testing batches of jets are considered.



(a) Development of loss over training epochs



(b) Normalized prediction error

Figure 10: Results of DNN jet momentum regression

1. particle p_t
2. particle p_t normalized to jet p_t
3. difference in η between particle and jet axis
4. difference in ϕ between particle and jet axis

DeepJetCore does not support ragged structures at the time of this project, meaning that the input shape must be fixed for all considered jets. As the number of constituents varies across different jets, the feature array for a single jet is padded with rows of 4 zeros. The input shape to the network is thus given by $(\text{jet batch size}, \text{maximal \# of constituents in any jet}, 4)$. Further note that the p_t of the jet constituents is passed unaltered through the network as these quantities are needed to evaluate the predicted jet p_t.
The network contains 3 layers, the first two of which are *GravNet* layers with parameters:

1. $\text{n\_propagate} = 64, \text{n\_dimensions} = 3, \text{n\_neighbours} = 15, \text{n\_filters} = 64$
2. $\text{n\_propagate} = 32, \text{n\_dimensions} = 3, \text{n\_neighbours} = 15, \text{n\_filters} = 32$

The final layer is a dense layer with 8 units and a linear activation function.
To compute the predicted jet p_t, the network output[7] of 8 values per particles is reduced to one final momentum correction factor by computing the mean. These correction factors then form the coefficients in the linear combination, leading to predicted jet p_t.

4.2.1 The GravNet layer

The GravNet layer is a distance-weighted graph network which has been developed by Shah Rukh Qasim et al, specifically for sparse input data such as the measurements of particles in a detector. I will only give a brief overview of the architecture of this network class, based on section 3 of [10].

[7] Actually, the networks returns 9 values per particle, with the last one being the particle p_t which is passed through the network unchanged.

The input to the GravNet layer is a 2D array containing a selection of F_{IN} features for every of the V particles. A dense layer then produces two sets of arrays, S and F_{LR}. The former array is of shape (V, n) and represents n coordinates per particle, allowing to map the input particles onto a graph in a $n-$dimensional space. The dimensionality of S is one of the parameters one can pass during the building of a GravNet layer through the argument *n_dimensions*. For each particle the array F_{LR} contains *n_propagate* values which can be interpreted as abstract particle features learned by the preceding dense layer. Every vertex in S is characterised by the associated *n_propagate* values in F_{LR}. For vertex j, the $i-th$ such feature is denoted by f_j^i. The graph representation of the input data allows to identify similar particles by considering the separation of the graph vertices. Complexity is added to the model by permitting the *n_neighbours* closest separated vertices[8] to affect each other. This is done through the following steps:

1. The connection between the pair of vertices (j, k) is quantified through

$$f_{jk}^i = f_j^i \times V(d_{jk})$$

   where the function $V(x) = \exp(-10|x|)$ acts as a potential[9], favouring closely separated vertices.

2. Based on the connection strengths, a new set of features $\tilde{f}_j{}^i$ for every vertex j is computed through so called gather function. As explained in section 3 of [10], it is advantageous to produce more than one set of new features. In the GravNet layers the following two gather functions are used:

   - $\tilde{f}_j{}^i = \max_k f_{jk}^i$, leading to the set of features $\tilde{F}_{LR}$
   - $\tilde{\tilde{f}}_j{}^i = \text{mean}_k\, f_{jk}^i$, leading to the set of features $\tilde{\tilde{F}}_{LR}$

Note that in steps 1 and 2, i indexes the features learned from the initial dense layer, while j runs over all vertices. Hence $\tilde{F}_{LR}, \tilde{\tilde{F}}_{LR}$ are of shape $(V, n\_propagate)$. Finally, the feature sets $F_{IN}, \tilde{F}_{LR}$, and$\tilde{\tilde{F}}_{LR}$ enter as the input to dense layer with *n_filters* units, producing the feature set F_{OUT} of shape $(V, n\_filters)$.
The strength of GravNet layers is that features of particle which are similar in the abstract space S are combined. This and the detector-geometry independence of the layers makes them a powerful addition to the DNN considered.

5 Acknowledgements

To conclude I would like to thank my supervisors for their guidance throughout this project and CERN for providing me with an excellent opportunity to experience world-leading research in such a stimulating environment. The Summer Student Program allowed me to further develop my research, problem solving, and organisation skills. Additionally, I developed several of technical skills including Git and debugging. Besides I also improved my general programming abilities and specifically my understanding of neural networks. Furthermore, the lecture program running parallel to the project work stimulated my interest in particle and collider physics, allowing me to extend on my physics understanding in numerous ways.

[8]The Euclidean distance is used to measure separation.
[9]Hence the name 'gravitational network layer'.

References

[1] Jesse Thaler. *Lecture 1 - Introduction to Jet*. MITP Summer School 2016: New Physics on Trial at LHC Run II. July 2016. URL: https://indico.mitp.uni-mainz.de/event/55/sessions/505/attachments/1483/1558/jthaler_MITP_Lecture1.pdf (visited on 23/06/2021).

[2] Matteo Cacciari. *Lecture 1 - Jet algorithms*. PRISMA Summer School. Sept. 2018. URL: https://indico.mitp.uni-mainz.de/event/167/attachments/651/689/Cacciari_1.pdf (visited on 20/08/2021).

[3] Noel Dawe et al. *scikit-hep/pyjet: Version 1.8.2*. Version 1.8.2. Jan. 2021. DOI: 10.5281/zenodo.4446849. URL: https://doi.org/10.5281/zenodo.4446849.

[4] Matteo Cacciari, Gavin P. Salam and Gregory Soyez. *FastJet user manual*. FastJet, June 2021. URL: http://fastjet.fr/repo/fastjet-doc-3.4.0.pdf (visited on 19/07/2021).

[5] Matteo Cacciari, Gavin P Salam and Gregory Soyez. "The anti-ktjet clustering algorithm". In: *Journal of High Energy Physics* 2008.04 (Apr. 2008), pp. 063–063. ISSN: 1029-8479. DOI: 10.1088/1126-6708/2008/04/063. URL: http://dx.doi.org/10.1088/1126-6708/2008/04/063.

[6] Rene Brun and Fons Rademakers. *TLorentzVector Class Reference*. Release 6.24/02. CERN, Aug. 2021. URL: https://root.cern.ch/doc/master/classTLorentzVector.html (visited on 05/08/2021).

[7] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*. Dec. 2013. URL: http://mpmath.org/ (visited on 12/08/2021).

[8] Julian Wack. *JulianWack/jet-momentum-regression-with- DeepJetCore: Trained model for jet momentum regression*. Version v1.0.0. Sept. 2021. DOI: 10.5281/zenodo.5414686. URL: https://doi.org/10.5281/zenodo.5414686.

[9] Jan Kieseler et al. *DeepJetCore*. Version 2.0. Feb. 2020. DOI: 10.5281/zenodo.3670882. URL: https://doi.org/10.5281/zenodo.3670882.

[10] Shah Rukh Qasim et al. "Learning representations of irregular particle-detector geometry with distance-weighted graph networks". In: *The European Physical Journal C* 79.7 (July 2019). ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-019-7113-9. URL: http://dx.doi.org/10.1140/epjc/s10052-019-7113-9.