1) Decision Tree Implementation :

- **How to run the code:** Decision tree is implemented in 'decision_tree.py' with command line arguments. To see the arguments supported give –'h' or '—help' option.
  Arguments supported:
  -i, --max_leaf_impurity : If internal node reaches below this impurity level, it'll be made as leaf (Default : 0)
  -l,--max_samples_in_leaf: If internal node gets below this samples size, it'll be made as leaf (Default: 5)
  -d, --min_pruning_node_depth : Minimum depth of node in tree to consider in post pruning. After this depth, if node is not giving any better accuracy, make it as leaf node (Default: 10)
  -k --kfold: K-Fold cross validation (Default: 1)
  -f, --impurity_func: Impurity fucntion (entropy, gini) (Default: gini)

  Example run:
  - *%run optimized_DT.py –i 0 –l 5 –d 10 –k 1 –f gini*
    accuracy: 0.8143

- **Cross validation :**
  To do cross validation pass the argument '-k' or '—kfold' with the number of folds
  - *%run optimized_DT.py -k 10*

  accuracy: 0.8046

- **Improvement Strategies :**
  Accuracy improvements:
  1) Pass the argument '—impurity_fun' or '-f' as 'gini' to use gini index instead of 'entropy'
  2) Post pruning: Post pruning is done after tree is constructed with prune data set. This strategy is implemented in post_prune method in code. In this code, we traverse the constructed tree with prune data and count the accuracies of each internal node when it acts as split or when it acts leaf node. Then we traverse one more time on tree and remove the internal nodes where accuracies are better when these are leaf nodes. Pass the parameter '–d 'or '--min_pruning_node_depth' to consider the all nodes after this depth for post pruning

  For multi way split, I have observed that accuracy doesn't improve much because we can always split the one multi-way split into multiple binary splits, so we can reduce the tree depth (but it may increase tree breadth) and it can make algorithm faster but it won't improve accuracy significantly compared to binary split tree.

Performance Improvements:

1) Split strategy : As our data set has real valued features, instead of splitting at each value of the feature to check for maximum information gain(or minimum impurity) , we first sort the feature values in increasing order and check impurity only at breakpoints where class labels are changing.
   For example, if (feature, class) vector is like this,
   {2.4, 0}, {2.6, 0}, {3.1, 1}, {3.2, 0} then split only at 3.1 and 3.2.

2) What's Cooking (Kaggle Challenge):

1) Feature Transformation:

As the dataset is huge and each ingredient is a list of words, this is considered as bag of words problem. So, for each cuisine sample, we combined the ingredients into one string and represented as vector in count matrix. This results in big sparse matrix for all ingredients words. For this 'CountVectorizer' package is used.

2) Decision Tree Classifier :

'DecisionTreeClassifier' module from sklearn package is used to classify count matrix using decision tree.

Accuracy for test set in Kaggle: 0.64119

3) KNN Classifier :

'KNeighborsClassifier' module from sklearn package is used to apply K-NN classifier on above count matrix

Accuracy for test set in Kaggle: 0.64360

Attached the code and test predictions for KNN, DT.