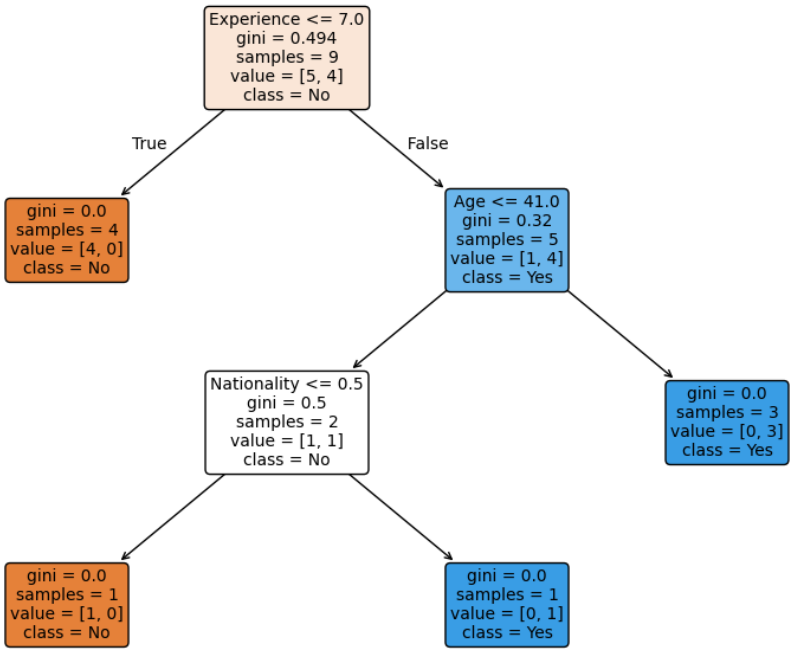
ds

November 24, 2024

[2]:
```
import pandas
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
import matplotlib.pyplot as plt

# Load the dataset
df = pandas.read_csv("TempData.csv")

# Map categorical data in the 'Nationality' column to numerical values
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)

# Map categorical data in the 'Go' column to numerical values
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

# Define the features (input variables) and target variable
features = ['Age', 'Experience', 'Rank', 'Nationality']  # Columns to be used␣
 ↪as features
X = df[features]  # Input features
y = df['Go']  # Target variable

# Initialize the decision tree classifier
dtree = DecisionTreeClassifier()

# Fit the decision tree classifier on the data
dtree = dtree.fit(X, y)

# Plot the decision tree
plt.figure(figsize=(12, 8))  # Set the size of the plot
plot_tree(
    dtree,
    feature_names=features,  # Display feature names
    class_names=['No', 'Yes'],  # Display class names for target variable
    filled=True,  # Fill nodes with colors representing the classes
    rounded=True,  # Use rounded corners for the boxes
    fontsize=10  # Set font size
)
```

```
plt.show()  # Show the plot

# Print a textual representation of the decision tree
print(export_text(dtree, feature_names=features))
```




```
|--- Experience <= 7.00
|   |--- class: 0
|--- Experience >  7.00
|   |--- Age <= 41.00
|   |   |--- Nationality <= 0.50
|   |   |   |--- class: 0
|   |   |--- Nationality >  0.50
|   |   |   |--- class: 1
|   |--- Age >  41.00
|   |   |--- class: 1
```

This cell loads a dataset, preprocesses categorical data, trains a decision tree classifier, and visualises the resulting model. The decision tree is used to predict the target variable (Go) based on features like Age, Experience, Rank, and Nationality.

[3]:
```
# Load the dataset
df = pandas.read_csv("poker-hand-training-true.csv")
```

```
# Define the features and target
features = ["S1", "R1", "S2", "R2", "S3", "R3", "S4", "R4", "S5", "R5"]
X_train = df[features]  # Features from training data
y_train = df["Class"]  # Target from training data

model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
print(f"Tree Depth: {model.get_depth()}")
print(f"Number of Leaves: {model.get_n_leaves()}")
```

```
Tree Depth: 30
Number of Leaves: 9196
```

0.1 Decision Tree Training Results

0.1.1 Tree Structure

- **Tree Depth**: 30
  - The depth of the tree is the longest path from the root node to a leaf node.
  - A depth of 30 indicates that the tree has grown very deep, creating a highly detailed set of rules for classification.
  - While this might improve training accuracy, it can lead to **overfitting**, where the model memorizes the training data instead of generalizing patterns.
- **Number of Leaves**: 9196
  - Leaves are the terminal nodes in the tree where predictions are made.
  - The tree has 9196 leaves, which suggests it has created a large number of distinct rules to classify the data.
  - While this level of complexity is expected for a dataset like Poker Hands (due to the many combinations of suits and ranks), it could also indicate **overfitting**.

0.1.2 Addressing Overfitting

To prevent overfitting and improve the model's generalization ability:

1. **Limit the Depth of the Tree**:
   - Restrict the maximum depth of the tree to control its complexity
2. **Limit the Minimum Samples per Leaf**:
   - Set a minimum number of samples required for a leaf node
3. **Prune the Tree**:
   - Use cost-complexity pruning (`ccp_alpha`) to remove overly specific splits

[3]:
```
model = DecisionTreeClassifier(random_state=42, min_samples_leaf=10, ␣
↪max_depth=10) # Added cap on depth and minimum samples per leaf
model.fit(X_train, y_train)
print(f"Tree Depth: {model.get_depth()}")
```

```
print(f"Number of Leaves: {model.get_n_leaves()}")
```

```
Tree Depth: 10
Number of Leaves: 684
```

[4]:
```
from sklearn.metrics import accuracy_score

# Get the effective alpha values for pruning
path = model.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

# Display alpha values and corresponding tree impurities
print("Effective Alphas:", ccp_alphas)
print("Impurities for each alpha:", impurities)
```

```
Effective Alphas: [0.00000000e+00 0.00000000e+00 3.99840064e-06 3.99840064e-06
 4.36189161e-06 6.05818279e-06 9.79694702e-06 1.19952019e-05
 1.19952019e-05 1.40706181e-05 1.46608023e-05 1.53877843e-05
 1.53885814e-05 1.56820389e-05 1.58302950e-05 1.59936026e-05
 1.59936026e-05 1.61147662e-05 1.72398573e-05 1.86592030e-05
 1.87822307e-05 1.91842455e-05 1.95178845e-05 1.99400759e-05
 2.20517853e-05 2.21278738e-05 2.22727308e-05 2.24152763e-05
 2.37900178e-05 2.39904038e-05 2.42327311e-05 2.54443677e-05
 2.62521254e-05 2.70194952e-05 2.82936934e-05 2.85768046e-05
 2.87884846e-05 3.04120776e-05 3.07901129e-05 3.10178959e-05
 3.11525045e-05 3.23506961e-05 3.29779824e-05 3.32547634e-05
 3.33532164e-05 3.34152053e-05 3.37195404e-05 3.38498419e-05
 3.39604418e-05 3.46082396e-05 3.46458819e-05 3.51076501e-05
 3.55740057e-05 3.59856058e-05 3.59856058e-05 3.61188858e-05
 3.63188058e-05 3.69083136e-05 3.71201634e-05 3.73597973e-05
 3.76964060e-05 3.83684910e-05 3.84067035e-05 3.86685153e-05
 3.89988932e-05 3.90726900e-05 3.91104063e-05 3.99037709e-05
 3.99049866e-05 3.99840064e-05 4.01963757e-05 4.02122449e-05
 4.02801842e-05 4.12787945e-05 4.13168066e-05 4.17991534e-05
 4.22118543e-05 4.25203996e-05 4.39316107e-05 4.43156071e-05
 4.50036436e-05 4.52366973e-05 4.53152073e-05 4.54632962e-05
 4.64399024e-05 4.69350721e-05 4.74254743e-05 4.78306051e-05
 4.79661615e-05 4.79808077e-05 4.83638536e-05 4.89814084e-05
 4.92110848e-05 4.92110848e-05 4.94802079e-05 4.95040079e-05
 4.95277259e-05 4.95801679e-05 4.95992079e-05 5.08931360e-05
 5.10218669e-05 5.14322421e-05 5.14714536e-05 5.18015016e-05
 5.19792083e-05 5.20718132e-05 5.22626447e-05 5.29139059e-05
 5.30190854e-05 5.30862924e-05 5.41222841e-05 5.42391739e-05
 5.43558800e-05 5.47478493e-05 5.50580079e-05 5.54417986e-05
 5.54915616e-05 5.55875211e-05 5.56929926e-05 5.63035509e-05
 5.66440091e-05 5.70334637e-05 5.73528413e-05 5.75175988e-05
 5.77364362e-05 5.79000723e-05 5.80653519e-05 5.82387366e-05
 5.82990027e-05 5.84052093e-05 5.86688402e-05 5.87774698e-05
```

5.89988705e-05 5.92634008e-05 5.94027362e-05 5.94458187e-05
5.95079694e-05 6.00127021e-05 6.01676561e-05 6.04928969e-05
6.05735100e-05 6.08571847e-05 6.11520098e-05 6.13941615e-05
6.16042720e-05 6.16185361e-05 6.17530765e-05 6.23650671e-05
6.25675656e-05 6.25895788e-05 6.25960442e-05 6.26446599e-05
6.28520522e-05 6.29087280e-05 6.32197212e-05 6.34984102e-05
6.37701920e-05 6.43024339e-05 6.43375129e-05 6.49740104e-05
6.50579756e-05 6.50879393e-05 6.51975862e-05 6.52215093e-05
6.53655566e-05 6.59458388e-05 6.59782200e-05 6.60782282e-05
6.62814740e-05 6.69500273e-05 6.69732107e-05 6.70786037e-05
6.71761610e-05 6.72239098e-05 6.72987744e-05 6.73430262e-05
6.78771969e-05 6.82951220e-05 6.83221356e-05 6.85883965e-05
6.92030880e-05 6.94573660e-05 6.96338006e-05 6.96374023e-05
7.02361520e-05 7.02840557e-05 7.13415601e-05 7.18883363e-05
7.21378115e-05 7.21677535e-05 7.24584820e-05 7.27658887e-05
7.27710544e-05 7.38752118e-05 7.38825349e-05 7.41130000e-05
7.42489198e-05 7.42779811e-05 7.53639193e-05 7.57839972e-05
7.59082915e-05 7.60721352e-05 7.62778279e-05 7.64083601e-05
7.66462475e-05 7.66475517e-05 7.72418305e-05 7.74747572e-05
7.78260125e-05 7.81346729e-05 7.82806379e-05 7.83504262e-05
7.88940046e-05 7.89016131e-05 7.89997391e-05 7.93295891e-05
7.93338579e-05 7.93414828e-05 7.93870486e-05 7.94464823e-05
7.98252128e-05 8.06561339e-05 8.07218591e-05 8.14078014e-05
8.16537680e-05 8.18231926e-05 8.21061950e-05 8.28557466e-05
8.29575517e-05 8.35563211e-05 8.44467464e-05 8.50136136e-05
8.55185598e-05 8.58679454e-05 8.63337717e-05 8.63826062e-05
8.66320139e-05 8.67029135e-05 8.68410139e-05 8.68962891e-05
8.70518333e-05 8.70954876e-05 8.72178601e-05 8.74520642e-05
8.76996790e-05 8.77692517e-05 8.86107184e-05 8.87091051e-05
8.88533475e-05 8.90853893e-05 8.92850801e-05 8.96120981e-05
8.96493908e-05 8.99029235e-05 9.00168672e-05 9.00211344e-05
9.02757639e-05 9.04065251e-05 9.04597730e-05 9.06189365e-05
9.09612572e-05 9.11764534e-05 9.15591677e-05 9.21218045e-05
9.42357671e-05 9.50557659e-05 9.56240563e-05 9.60934308e-05
9.62146904e-05 9.66718431e-05 9.70249944e-05 9.75065429e-05
9.76289646e-05 9.79125105e-05 9.82607696e-05 9.87853859e-05
9.93232554e-05 9.93785931e-05 9.94024159e-05 9.94604469e-05
9.94744959e-05 9.95785486e-05 1.00404283e-04 1.01476187e-04
1.01753317e-04 1.02866746e-04 1.02879482e-04 1.02906823e-04
1.02985018e-04 1.03083416e-04 1.03536301e-04 1.03630520e-04
1.04410489e-04 1.04550763e-04 1.04735590e-04 1.04951432e-04
1.05998513e-04 1.06160333e-04 1.06369555e-04 1.06458214e-04
1.06655777e-04 1.06814417e-04 1.07288741e-04 1.07766417e-04
1.08195175e-04 1.08803483e-04 1.08907754e-04 1.08982560e-04
1.09231579e-04 1.09378030e-04 1.09640354e-04 1.09869668e-04
1.10279698e-04 1.10676409e-04 1.10678749e-04 1.10961123e-04
1.11054996e-04 1.11329322e-04 1.11344001e-04 1.12942417e-04
1.13574417e-04 1.13785714e-04 1.14365196e-04 1.14703483e-04

```
 1.14954018e-04 1.15062787e-04 1.15324657e-04 1.16027069e-04
 1.16610900e-04 1.16737294e-04 1.17280019e-04 1.17563290e-04
 1.17648278e-04 1.18127975e-04 1.18270152e-04 1.18324775e-04
 1.18449084e-04 1.19097822e-04 1.19952019e-04 1.20190689e-04
 1.22201985e-04 1.22629520e-04 1.22789157e-04 1.23003210e-04
 1.23909165e-04 1.25697952e-04 1.26068813e-04 1.26236158e-04
 1.26382739e-04 1.27016236e-04 1.27154239e-04 1.28113766e-04
 1.28446140e-04 1.28680148e-04 1.29615822e-04 1.29901495e-04
 1.30744584e-04 1.30801440e-04 1.31284072e-04 1.32281241e-04
 1.33070977e-04 1.33412833e-04 1.33600086e-04 1.33795693e-04
 1.34725996e-04 1.35376766e-04 1.36288733e-04 1.38501011e-04
 1.41430742e-04 1.43562485e-04 1.43975536e-04 1.44565970e-04
 1.44684606e-04 1.45290789e-04 1.45553323e-04 1.45712393e-04
 1.46100023e-04 1.46234670e-04 1.47674623e-04 1.48186602e-04
 1.49056466e-04 1.50852708e-04 1.53229588e-04 1.53239521e-04
 1.53355333e-04 1.55831983e-04 1.55896887e-04 1.55908334e-04
 1.55934757e-04 1.56105004e-04 1.57226358e-04 1.59761741e-04
 1.59979615e-04 1.60030017e-04 1.60085426e-04 1.61353124e-04
 1.61504026e-04 1.63010804e-04 1.65249472e-04 1.67939288e-04
 1.68660124e-04 1.71366177e-04 1.71520040e-04 1.71774427e-04
 1.72304116e-04 1.72576156e-04 1.73548762e-04 1.80277387e-04
 1.83090029e-04 1.83729220e-04 1.86187937e-04 1.86530584e-04
 1.86803585e-04 1.89578884e-04 1.89658713e-04 1.89905326e-04
 1.91543396e-04 1.94372860e-04 1.96519131e-04 2.01768260e-04
 2.04804841e-04 2.05548526e-04 2.08512480e-04 2.11449824e-04
 2.15000993e-04 2.17040621e-04 2.22863530e-04 2.27764158e-04
 2.31741583e-04 2.32012161e-04 2.33483678e-04 2.39751287e-04
 2.40790792e-04 2.50388195e-04 2.55272897e-04 2.67932955e-04
 2.79027485e-04 2.85590715e-04 2.85826741e-04 2.88095353e-04
 3.15464124e-04 3.32072104e-04 3.43465919e-04 3.51770294e-04
 3.58094046e-04 3.59515787e-04 3.76798757e-04 3.81341428e-04
 4.36344364e-04]
Impurities for each alpha: [0.48998738 0.48998738 0.48999138 0.48999538
0.48999974 0.4900058
 0.49001559 0.49002759 0.49003959 0.49005366 0.49006832 0.4900837
 0.49009909 0.49011477 0.49013061 0.4901466  0.49016259 0.49017871
 0.49019595 0.49021461 0.49023339 0.49025257 0.49027209 0.49029203
 0.49031408 0.49033621 0.49035848 0.4903809  0.49040469 0.49042868
 0.49045291 0.49047836 0.49050461 0.49053163 0.49055992 0.4905885
 0.49061729 0.4906477  0.49067849 0.49070951 0.49074066 0.49080536
 0.49083834 0.49087159 0.49090495 0.49093836 0.49097208 0.49103978
 0.49107374 0.49110835 0.491143   0.4911781  0.49121368 0.49124966
 0.49128565 0.49132177 0.49135809 0.49139499 0.49146923 0.49150659
 0.49154429 0.49158266 0.49162107 0.49165973 0.49169873 0.49173781
 0.49177692 0.49181682 0.49185673 0.49189671 0.4919771  0.49205753
 0.49209781 0.49213909 0.4921804  0.4922222  0.49226441 0.49230693
 0.49235087 0.49239518 0.49244018 0.49248542 0.49253074 0.4925762
 0.49266908 0.49271601 0.49276344 0.49281127 0.49285924 0.49290722
```

0.49295558 0.49300456 0.49305377 0.49310299 0.49315247 0.49320197
0.49330102 0.4933506 0.4934498 0.49355159 0.49360261 0.49375691
0.49380838 0.49391198 0.49396396 0.49401603 0.4940683 0.49412121
0.49417423 0.49422732 0.49438968 0.49449816 0.49460687 0.49471637
0.49482648 0.49488193 0.49493742 0.49499301 0.4950487 0.495105
0.49516165 0.49521868 0.49527603 0.49533355 0.49539129 0.49544919
0.49550725 0.49556549 0.49562379 0.49568219 0.49574086 0.49585842
0.49591742 0.49597668 0.49603608 0.49609553 0.49615504 0.49627506
0.49633523 0.49639572 0.4964563 0.49651715 0.49657831 0.4966397
0.4967013 0.49682454 0.49688629 0.49694866 0.49713636 0.49719895
0.49726155 0.49732419 0.49738704 0.49744995 0.49751317 0.49757667
0.49764044 0.49770474 0.49776908 0.49783406 0.49796417 0.49802926
0.49809446 0.49815968 0.49822504 0.49848883 0.49855481 0.49862088
0.49875345 0.4988204 0.49888737 0.49895445 0.49902162 0.49915607
0.49922337 0.49929071 0.49935859 0.49942689 0.49949521 0.4995638
0.499633   0.49984137 0.49998064 0.50005028 0.50026099 0.50033127
0.50040261 0.50061828 0.50069041 0.50076258 0.5009075 0.50105303
0.5011258 0.50119968 0.50127356 0.50142179 0.50149603 0.50164459
0.50171995 0.50187152 0.50194743 0.5020235 0.50217606 0.50225247
0.50232911 0.50240576 0.502483   0.50263795 0.50271578 0.50279391
0.50287219 0.50295054 0.50302944 0.50318724 0.50326624 0.5034249
0.50350423 0.50358358 0.50366296 0.50374241 0.50382223 0.50390289
0.50398361 0.50414643 0.50422808 0.5043099 0.50439201 0.50447487
0.50455782 0.50464138 0.50489472 0.50506475 0.50515027 0.50523613
0.50532247 0.50549523 0.50558187 0.50566857 0.50575541 0.50584231
0.50601641 0.5061035 0.50619072 0.50636563 0.50662873 0.50671649
0.50680511 0.50689381 0.50698267 0.50707175 0.50716104 0.5076091
0.50769875 0.50778865 0.50787867 0.50796869 0.50832979 0.5084202
0.50851066 0.50860128 0.50869224 0.50878342 0.50905809 0.50933446
0.50942869 0.50952375 0.50981062 0.50990672 0.51000293 0.51038962
0.51048664 0.51058415 0.51068178 0.5108776 0.51097586 0.51117343
0.51127276 0.51137214 0.51147154 0.511571   0.51167047 0.51186963
0.51197004 0.51217299 0.51247825 0.51268398 0.51299262 0.51319843
0.51330142 0.5134045 0.51371511 0.51392237 0.51402678 0.51496774
0.51507247 0.51517743 0.51538942 0.51549558 0.51570832 0.51592124
0.51602789 0.51613471 0.51634929 0.51645705 0.51656525 0.51667405
0.51678296 0.51700092 0.51721939 0.51732877 0.51754805 0.51765792
0.51798875 0.51821011 0.51843147 0.51854243 0.51865348 0.51887614
0.51909883 0.51921177 0.51932534 0.51955292 0.51978165 0.51989635
0.5200113 0.52024143 0.52047208 0.52058811 0.52093794 0.52105468
0.52117196 0.52152465 0.52164229 0.52187855 0.52199682 0.52235179
0.52258869 0.52270779 0.52282774 0.52294793 0.52307014 0.52319276
0.52331555 0.52343856 0.52356247 0.52368816 0.52381423 0.52444541
0.52520371 0.52571178 0.52622039 0.52634851 0.52647695 0.52660563
0.52673525 0.52686515 0.52699589 0.5273883 0.52765087 0.52778315
0.52818236 0.52831577 0.52871657 0.52885037 0.5289851 0.52912047
0.52939305 0.52967005 0.52995291 0.53009648 0.53024045 0.53038502
0.53067439 0.53081968 0.53125634 0.53140205 0.53154815 0.53184062

```
0.53198829 0.53213648 0.53243459 0.53349056 0.53425671 0.53440995
0.53456331 0.5354983  0.53581009 0.535966   0.53612193 0.53659025
0.53706193 0.53722169 0.53738167 0.539142   0.53946217 0.53994623
0.54010773 0.54027074 0.54060124 0.54076918 0.5411065  0.54127787
0.54144939 0.54162116 0.54179347 0.54248377 0.54265732 0.5428376
0.54302069 0.54320442 0.54339061 0.54357714 0.54395074 0.54414032
0.54432998 0.54451989 0.54471143 0.54510018 0.54549321 0.54569498
0.54610459 0.54631014 0.54651865 0.54694155 0.54715655 0.54737359
0.54826505 0.54940387 0.55033084 0.55079486 0.55149531 0.55173506
0.55269822 0.553199   0.55422009 0.55529182 0.55696599 0.55725158
0.55782323 0.55811133 0.55842679 0.55942301 0.55976648 0.56011825
0.56047634 0.56299295 0.56336975 0.56375109 0.56811453]
```

0.2 Cost Complexity Pruning Results

0.2.1 Effective Alphas (`ccp_alpha`)

- **Definition**:
  - The `ccp_alpha` values represent thresholds for pruning the decision tree.
  - Each alpha controls the pruning strength, where larger values prune more nodes, simplifying the tree.
- **Values**:
  - Example values from the dataset:
    `[0.00000000e+00, 2.13248034e-05, 2.33240037e-05, ...,`
    `3.76798757e-04, 3.81341428e-04, 4.36344364e-04]`
  - The range begins at `0.00000000e+00` (no pruning) and increases to `4.36344364e-04` (high pruning).

0.2.2 Impurities for Each Alpha

- **Definition**:
  - Impurities represent the total impurity (e.g., Gini) of the tree for a given `ccp_alpha`.
  - As pruning increases, the impurity rises because the tree becomes simpler and less specialized.
- **Values**:
  - Example impurity values:
    `[0.00000000e+00, 6.39744102e-05, 1.33946421e-04, ...,`
    `5.63369749e-01, 5.63751091e-01, 5.68114534e-01]`
  - At `ccp_alpha = 0.00000000e+00`, the tree is fully grown with no impurity.
  - As `ccp_alpha` increases, the impurity grows, reflecting the tree's simplification.

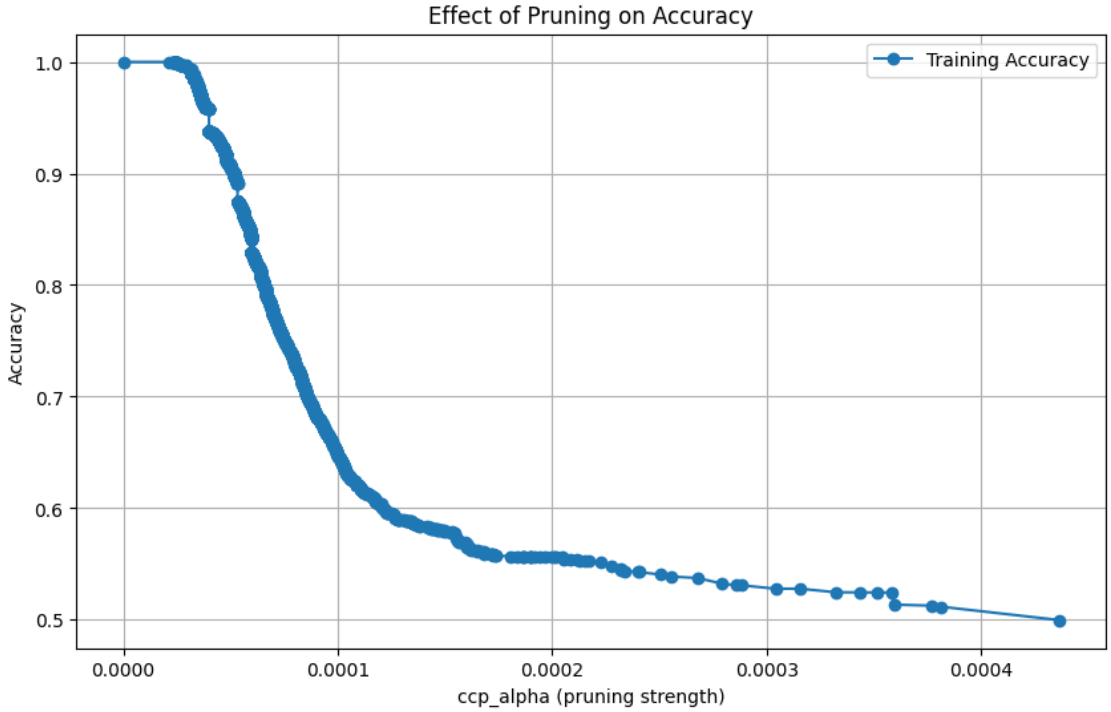
0.2.3 Interpreting the Results

1. **Alpha Values (`ccp_alpha`)**:
   - **Low Alpha (`0.00000000e+00`)**:
     - The tree is fully grown, with no pruning applied.
     - It likely overfits the training data, resulting in perfect accuracy but poor generalization.

- **Moderate Alpha (`2.13248034e-05` to `3.81341428e-04`)**:
  - Nodes with minimal contributions to impurity reduction are pruned.
  - The tree becomes simpler, balancing training accuracy and generalization.
- **High Alpha (`4.36344364e-04`)**:
  - The tree is heavily pruned, with fewer nodes and high impurity.
  - It may underfit the data, leading to poor performance on both training and test sets.

2. **Impurity Values**:
   - **Low Impurity (`0.00000000e+00`)**:
     - At low `ccp_alpha`, the tree is fully grown and fits the training data perfectly.
   - **High Impurity (`5.68114534e-01`)**:
     - At high `ccp_alpha`, the tree is simplified to the point of underfitting, where it lacks the complexity needed to represent the data accurately.

[16]:
```
# Train models for each alpha
models = []
train_scores = []

for alpha in ccp_alphas:
    # Train a new tree for each alpha
    pruned_model = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
    pruned_model.fit(X_train, y_train)
    models.append(pruned_model)

    # Evaluate on training
    train_scores.append(pruned_model.score(X_train, y_train))

# Plot accuracy vs. alpha
plt.figure(figsize=(10, 6))
plt.plot(ccp_alphas, train_scores, label="Training Accuracy", marker='o')
plt.xlabel("ccp_alpha (pruning strength)")
plt.ylabel("Accuracy")
plt.title("Effect of Pruning on Accuracy")
plt.legend()
plt.grid()
plt.show()
```



0.3 Understanding `ccp_alpha` Values for Pruning

0.3.1 What is `ccp_alpha`?

- `ccp_alpha`: A hyperparameter that determines the strength of pruning applied to the decision tree.
- Higher values of `ccp_alpha` prune more nodes from the tree, simplifying its structure.

0.3.2 What These Numbers Mean

Each value in the array represents a pruning threshold (alpha). As `ccp_alpha` increases: 1. The decision tree removes nodes that add less value to reducing impurity. 2. The tree becomes progressively smaller and simpler.

Example of alpha values:

4.36189161e-06, 6.05818279e-06, 9.79694702e-06, 1.19952019e-05, 1.19952019e-05, 1.40706181e-05, 1.46608023e-05, 1.53877843e-05, 1.53885814e-05, 1.56820389e-05, 1.58302950e-05, 1.59936026e-05, 1.59936026e-05, 1.61147662e-05, 1.72398573e-05, 1.86592030e-05, 1.87822307e-05, 1.91842455e-05, 1.95178845e-05, 1.99400759e-05

Interpreting the Example Set

- **Range**: The values start very small and gradually increase, indicating incremental steps for pruning strength.

- **Magnitude**: The values are small because they represent minimal changes in impurity reduction at each pruning step.

0.3.3 Why These Values Matter

1. **Tree Impurity vs. Complexity**: Each alpha value corresponds to a tree with a specific impurity level and number of nodes.
2. **Choosing the Optimal Alpha**: The goal is to find the **smallest alpha** that maximizes test accuracy without overfitting the training data.

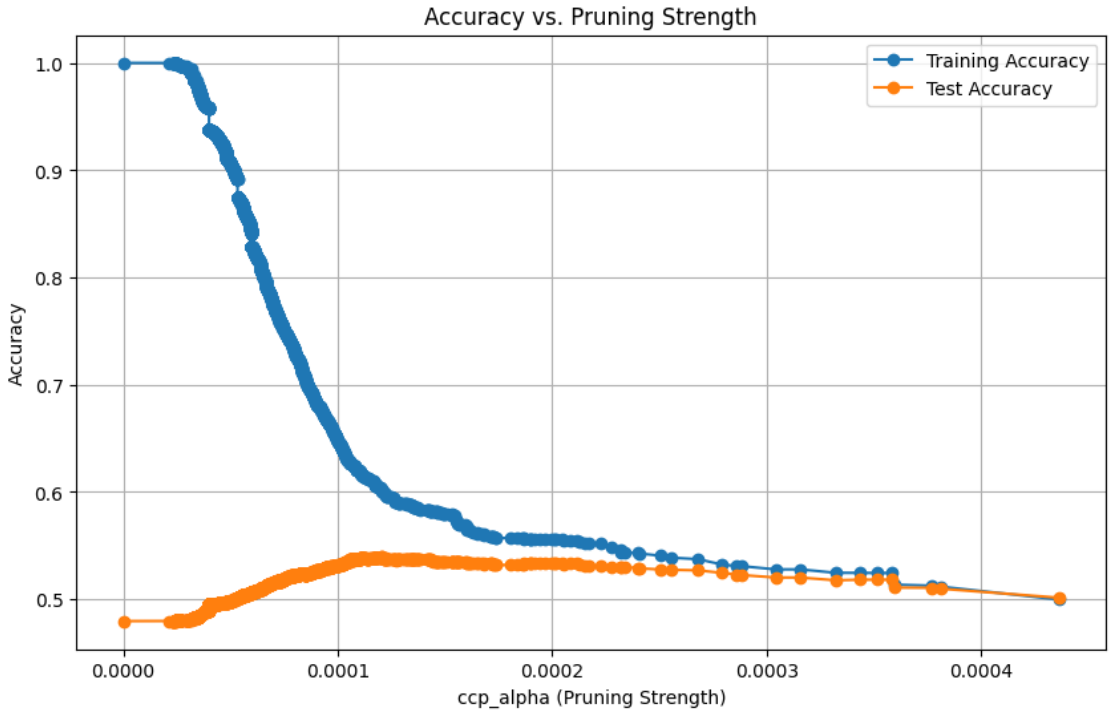
0.3.4 Observations from the Pruning Plot

The pruning plot with training accuracy only shows a line starting at 1.0, slowly declining, then dropping sharply to around 0.6, and then gradually descending to 0.5. Here's what this means:

- **Initial High Accuracy**: At very low `ccp_alpha` values, the tree is highly complex and overfitted, achieving nearly perfect accuracy on training data.
- **Sharp Drop**: As `ccp_alpha` increases, significant pruning begins, and the model starts losing accuracy on the training set, indicating it's simplifying and removing overfitted details.
- **Gradual Decline to 0.5**: At high `ccp_alpha` values, the tree becomes very shallow, resulting in underfitting, where it lacks complexity to capture patterns in the training data.

[20]:
```
# Load test data
test_data = pandas.read_csv("poker-hand-testing.csv")

# Define features and target for test data
X_test = test_data[features]
y_test = test_data["Class"]

# Initialize lists for storing accuracies
train_scores = []
test_scores = []

# Train and evaluate the tree for each alpha
for alpha in ccp_alphas:
    pruned_model = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
    pruned_model.fit(X_train, y_train)

    # Record training and test accuracy
    train_scores.append(pruned_model.score(X_train, y_train))
    test_scores.append(pruned_model.score(X_test, y_test))

    # Plot training and test accuracy vs. pruning strength
plt.figure(figsize=(10, 6))
plt.plot(ccp_alphas, train_scores, label="Training Accuracy", marker='o')
plt.plot(ccp_alphas, test_scores, label="Test Accuracy", marker='o')
plt.xlabel("ccp_alpha (Pruning Strength)")
plt.ylabel("Accuracy")
plt.title("Accuracy vs. Pruning Strength")
```

```
plt.legend()
plt.grid()
plt.show()
```



0.4 Pruning Results and Analysis

0.4.1 Observations

1. **Initial High Training Accuracy (1.0)**:
   - When the tree is fully grown (`ccp_alpha = 0`), it perfectly fits the training data.
   - This leads to overfitting, where the model memorizes the training data but cannot generalize to unseen test data.
   - Test accuracy is low (less than 0.5) at this stage.
2. **Peak Test Accuracy (~0.53)**:
   - As pruning strength (`ccp_alpha`) increases, the tree simplifies and begins generalizing better.
   - Test accuracy peaks around a `ccp_alpha` value where the model achieves a good balance between complexity and generalization.
3. **Declining Test Accuracy**:
   - Beyond the optimal pruning point, further pruning makes the tree too simple.
   - Both training and test accuracies converge around 0.5, indicating the model is underfitting and cannot capture meaningful patterns.

0.4.2 Interpretation

1. **Overfitting**:
   - The initial high training accuracy (1.0) and low test accuracy (<0.5) are signs of overfitting.
   - The fully grown tree memorizes the training data, failing to generalize to test data.
2. **Optimal Pruning Strength**:
   - The `ccp_alpha` value where **test accuracy peaks (~0.53)** represents the best pruning strength.
   - At this point, the tree achieves the best balance between fitting the training data and generalizing to unseen data.
3. **Underfitting**:
   - As pruning becomes too aggressive (high `ccp_alpha`), the tree loses complexity and underfits.
   - Training and test accuracies converge (0.5), indicating the tree is too simple to make meaningful predictions.

0.4.3 Conclusion

- The optimal pruning strength (`ccp_alpha`) provides the best balance between simplicity and generalization.
- By selecting this pruning strength, the decision tree avoids overfitting and underfitting, resulting in improved test accuracy.

[23]:
```
# Select the optimal pruning strength
optimal_alpha = ccp_alphas[test_scores.index(max(test_scores))]
print(f"Optimal ccp_alpha: {optimal_alpha}")

# Train the final pruned model
final_model = DecisionTreeClassifier(random_state=42, ccp_alpha=optimal_alpha)
final_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = final_model.predict(X_test)

# Evaluate the pruned model
from sklearn.metrics import accuracy_score, classification_report
print("Accuracy on Test Data:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Optimal ccp_alpha: 0.0001201906888875664
Accuracy on Test Data: 0.539267
```

```
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.57      0.70      0.63    501209
           1       0.49      0.45      0.47    422498
           2       0.25      0.03      0.05     47622
           3       0.47      0.00      0.01     21121
           4       0.00      0.00      0.00      3885
           5       0.99      0.23      0.38      1996
           6       0.00      0.00      0.00      1424
           7       0.00      0.00      0.00       230
           8       0.00      0.00      0.00        12
           9       0.00      0.00      0.00         3

    accuracy                           0.54   1000000
   macro avg       0.28      0.14      0.15   1000000
weighted avg       0.52      0.54      0.52   1000000
```

```
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

0.5 Pruning Results and Analysis

0.5.1 Current Performance

- **Optimal `ccp_alpha`**: `0.0001201906888875664`
  - This pruning strength was selected as it balances model complexity and generalization.
  - The tree was pruned using this value, simplifying its structure while attempting to retain its predictive ability.
- **Accuracy on Test Data**: `0.539267` (53.93%)
  - The model correctly classifies approximately **53.93%** of the test data.
  - This is above random guessing but indicates the model struggles to generalize effectively, particularly for rare poker hands.

0.5.2 Classification Report

The classification report evaluates the model's performance on each poker hand category:

Class	Poker Hand	Precision	Recall	F1-Score	Support
0	Nothing	0.57	0.70	0.63	501209
1	One Pair	0.49	0.45	0.47	422498
2	Two Pair	0.25	0.03	0.05	47622
3	Three of a Kind	0.47	0.00	0.01	21121
4	Straight	0.00	0.00	0.00	3885
5	Flush	0.99	0.23	0.38	1996
6	Full House	0.00	0.00	0.00	1424
7	Four of a Kind	0.00	0.00	0.00	230
8	Straight Flush	0.00	0.00	0.00	12
9	Royal Flush	0.00	0.00	0.00	3

Key Observations: 1. **Class Imbalance**: - The dataset is heavily imbalanced, with most samples belonging to class `0` (Nothing) and far fewer in classes like `9` (Royal Flush) and `8` (Straight Flush). - This imbalance skews the model's predictions heavily toward the majority classes.

2. **Performance on Majority Classes**:
   - Classes `0` (Nothing) and `1` (One Pair) have better precision, recall, and F1-scores due to their dominance in the dataset.
3. **Poor Performance on Rare Classes**:
   - Rare classes like `8` (Straight Flush) and `9` (Royal Flush) are not predicted correctly, as indicated by precision, recall, and F1-scores of `0.00`.

0.5.3 Current Problem

- The current training set contains **25,000 hands**, while the test set contains **1,000,000 hands**.
- This large discrepancy between the training and test data limits the model's ability to learn meaningful patterns, especially for rare poker hands like Royal Flush or Straight Flush.
- A small training set increases the likelihood of overfitting, where the model memorizes specific patterns in the training data rather than generalizing.

0.5.4 Plan: Increase Training Set Size

- To improve the model's performance, the first step is to increase the training set size from **25,000 hands** to **50,000–100,000 hands**.
- This can be achieved by re-splitting the data and allocating a larger portion of the dataset to the training set.

0.5.5 Advantages of Increasing the Training Set

1. **Learn More Patterns**:

- A larger training set provides the model with more examples of each poker hand, improving its ability to identify and generalize patterns, especially for rare classes.

2. **Reduce Overfitting**:
   - With more training data, the model relies less on memorizing specific examples and learns broader, generalizable decision rules.
3. **Improve Accuracy**:
   - Increasing the size of the training set allows the model to make more accurate predictions on the test data, as it has been exposed to a greater variety of poker hands.
4. **Better Representation of Rare Hands**:
   - Rare poker hands like Royal Flush (class 9) or Straight Flush (class 8) are more likely to appear in a larger training set, improving the model's ability to correctly classify them.

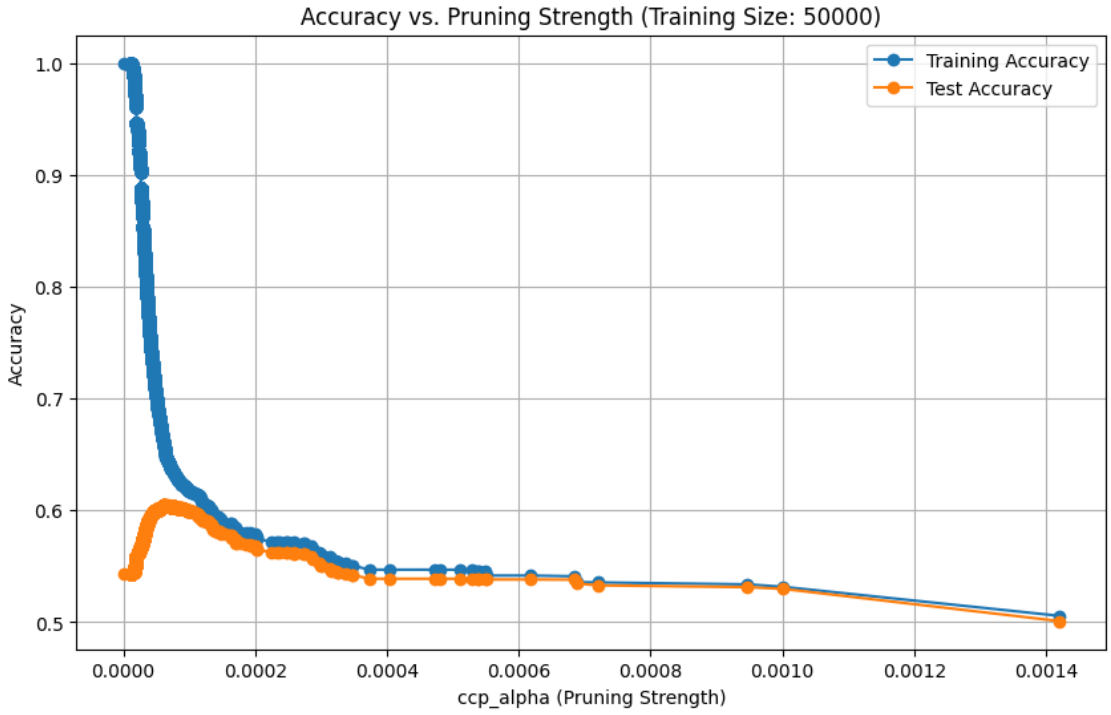
[ ]:
```
from sklearn.model_selection import train_test_split  # Import train_test_split␣
↪for splitting data into training and test sets

# Dictionary to store datasets for different training sizes
datasets = {}

# Combine current training and test datasets
combined_data = pandas.concat([df, test_data])  # Merge the existing training␣
↪and test datasets into one
combined_features = combined_data[["S1", "R1", "S2", "R2", "S3", "R3", "S4",␣
↪"R4", "S5", "R5"]]  # Extract feature columns (suits and ranks)
combined_target = combined_data["Class"]  # Extract the target column (poker␣
↪hand classes)

# Define a function to create training and test sets based on a specified␣
↪training size
def create_train_test(combined_features, combined_target, training_size):
    total_samples = combined_features.shape[0]  # Total number of samples in␣
↪the dataset
    train_ratio = training_size / total_samples  # Calculate the ratio of␣
↪training samples to total samples

    # Split the dataset into training and test sets based on the calculated␣
↪ratio
    X_train, X_test, y_train, y_test = train_test_split(
        combined_features, combined_target, test_size=(1 - train_ratio),␣
↪random_state=42
    )

    # Print the sizes of the training and test sets
    print(f"Training Set Size: {X_train.shape[0]}")
    print(f"Test Set Size: {X_test.shape[0]}")
```

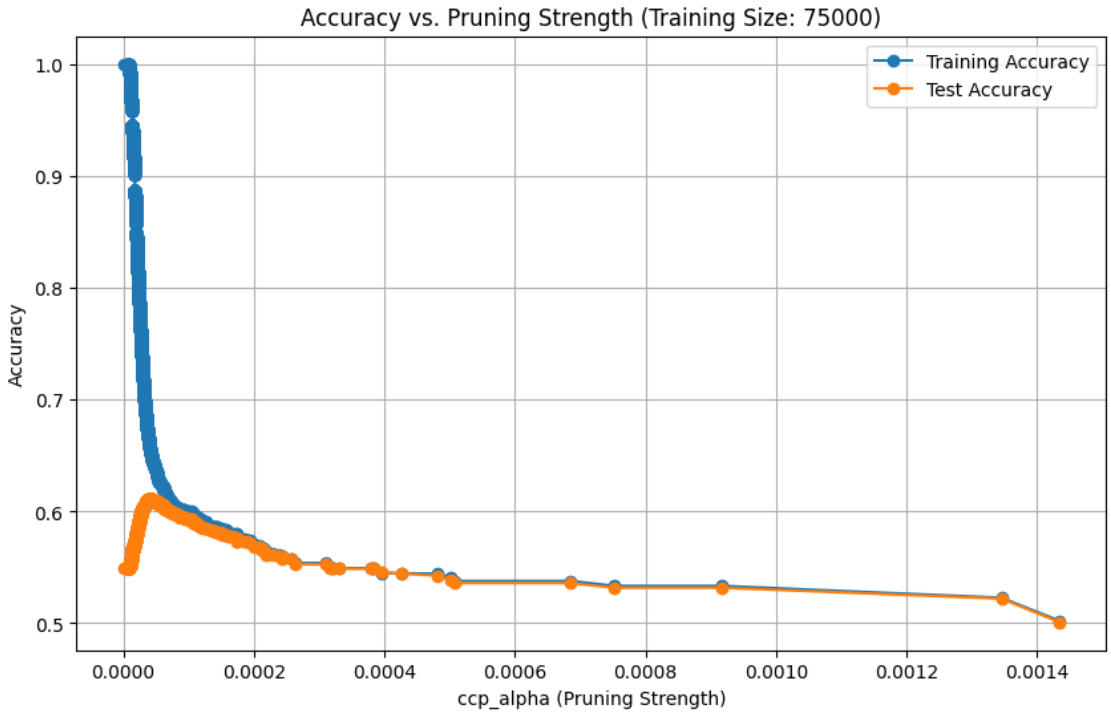
```
    return X_train, X_test, y_train, y_test  # Return the split datasets

# Define a list of training sizes to evaluate: 50k, 75k, and 100k
training_sizes = [50000, 75000, 100000]  # Specify desired training sizes

# Loop through the specified training sizes to create datasets and print results
for size in training_sizes:
    print(f"\nFor Training Size: {size}")  # Indicate the current training size␣
 ↪being processed
    X_train, X_test, y_train, y_test = create_train_test(combined_features,␣
 ↪combined_target, size)  # Generate the split datasets

    # Store the datasets in the dictionary with the training size as the key
    datasets[size] = (X_train, X_test, y_train, y_test)
```

```
For Training Size: 50000
Training Set Size: 50000
Test Set Size: 975010

For Training Size: 75000
Training Set Size: 75000
Test Set Size: 950010

For Training Size: 100000
Training Set Size: 100000
Test Set Size: 925010
```

[ ]:
```
# Loop through stored datasets and plot graphs
for size, (X_train, X_test, y_train, y_test) in datasets.items():
    # Get effective alpha values for pruning
    pruned_model = DecisionTreeClassifier(random_state=42)
    path = pruned_model.cost_complexity_pruning_path(X_train, y_train)
    ccp_alphas = path.ccp_alphas  # Extract alpha values

    # Initialize lists for accuracies
    train_scores = []
    test_scores = []

    # Train and evaluate the tree for each alpha
    for alpha in ccp_alphas:
        pruned_model = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
        pruned_model.fit(X_train, y_train)

        # Record accuracies for training and test sets
        train_scores.append(pruned_model.score(X_train, y_train))
        test_scores.append(pruned_model.score(X_test, y_test))
```

```
# Plot training and test accuracy for the current training size
plt.figure(figsize=(10, 6))
plt.plot(ccp_alphas, train_scores, label="Training Accuracy", marker='o')
plt.plot(ccp_alphas, test_scores, label="Test Accuracy", marker='o')
plt.xlabel("ccp_alpha (Pruning Strength)")
plt.ylabel("Accuracy")
plt.title(f"Accuracy vs. Pruning Strength (Training Size: {size})")
plt.legend()
plt.grid()
plt.show()
```





0.6 Pruning Results and Analysis

0.6.1 Observations

1. **Increase in Accuracy**

- In both using 50,000 hands and 75,000 hands in the training set increased the accuracy of the model with the tes data to over 6.0.
- This is a good increase however is not high enough to be in the desired accuracy range for the model,

2. **Unrealistic Pruning Time**

- The times it takes the prune the increased training set takes to long to efficiently work with and see results.
- The training set with 50,000 hands of poker tooke 100 minutes, the 75,000 took over 300 minutes and the 100,000 hands took too long to finish.
- Along with VS code I tried Google Colabs and similar results were seen. It doesn't seem to be a limit on my laptop with CPU, GPU or RAM it more seems that vs code will not use more than 400mb and there doesn't seem to be a work around

3. **Will need SMOTE**

- As increasing the data set does not seem realistic, SMOTE(Synthetic Minority Oversampling Technique) is going to be implemented.
- SMOTE will allow for increased accuracy on minority classes such as Royal FLush.

- **0.7 This is done by duplicating the minority examples in the training model. This can balance the class distribution but does not provide any additional information to the model.**

0.7.1 Interpretation

1. **Efficiency Challenges**

- Pruning larger datasets, such as 75,000 and 100,000 hands, results in excessive computation times, exceeding practical limits for iterative testing.
- This inefficiency indicates a need for optimised algorithms or alternative approaches to manage computational constraints.

2. **Class Imbalance Issues**

- Minority classes, such as "Royal Flush," remain poorly predicted despite pruning.
- Class imbalance affects the model's ability to generalise, necessitating strategies like SMOTE for better representation of underrepresented classes.

3. **Pruning Impact**

- Moderate pruning (lower ccp_alpha) improves generalisation, with a visible increase in test accuracy (~6.0).

- **0.8 However, aggressive pruning (higher ccp_alpha) causes underfitting, reducing both training and test accuracy.**

0.8.1 Conclusion

- Moderate pruning enhances the model's ability to generalise by mitigating overfitting without oversimplifying.
- Addressing class imbalance with SMOTE can complement pruning by improving accuracy for minority classes.
- Computational constraints highlight the importance of scalable solutions or more efficient frameworks for processing large datasets.

[7]:
```
# Loop through stored datasets and plot graphs
for size, (X_train, X_test, y_train, y_test) in datasets.items():
    # Get effective alpha values for pruning
    pruned_model = DecisionTreeClassifier(random_state=42)
    path = pruned_model.cost_complexity_pruning_path(X_train, y_train)
    ccp_alphas = path.ccp_alphas  # Extract alpha values

    # Initialize lists for accuracies
    train_scores = []
    test_scores = []
```

```
    # Train and evaluate the tree for each alpha
    for alpha in ccp_alphas:
        pruned_model = DecisionTreeClassifier(random_state=42, ccp_alpha=alpha)
        pruned_model.fit(X_train, y_train)

        # Record accuracies for training and test sets
        train_scores.append(pruned_model.score(X_train, y_train))
        test_scores.append(pruned_model.score(X_test, y_test))
```

[8]:
```
# Select the optimal pruning strength
optimal_alpha = ccp_alphas[test_scores.index(max(test_scores))]
print(f"Optimal ccp_alpha: {optimal_alpha}")

# Train the final pruned model
final_model = DecisionTreeClassifier(random_state=42, ccp_alpha=optimal_alpha)
final_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = final_model.predict(X_test)

# Evaluate the pruned model
from sklearn.metrics import accuracy_score, classification_report
print("Accuracy on Test Data:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Optimal ccp_alpha: 6.274047417807971e-05
Accuracy on Test Data: 0.6048481554035343
```

```
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.63      0.77      0.69    488420
           1       0.56      0.51      0.54    412201
           2       0.34      0.03      0.06     46475
           3       0.57      0.06      0.11     20564
           4       0.00      0.00      0.00      3775
           5       0.00      0.00      0.00      1942
           6       0.00      0.00      0.00      1386
           7       0.00      0.00      0.00       225
           8       0.00      0.00      0.00        14
           9       0.00      0.00      0.00         8

    accuracy                           0.60    975010
   macro avg       0.21      0.14      0.14    975010
weighted avg       0.58      0.60      0.58    975010
```

0.8.2 Current Performance

- **Training Set Size**: 50,000 hands
  - A larger training set was used to enhance model learning, especially for underrepresented classes.
- **Accuracy**: 60%
  - The model's accuracy has improved from the initial 54% to 60%, indicating better performance in correctly classifying the data. However, the improvement is still limited, particularly for minority classes.

0.8.3 Classification Report

The updated classification report presents the performance of the decision tree classifier for each class:

Class	Category	Precision	Recall	F1-Score	Support
0	Class 0	0.63	0.77	0.69	488420
1	Class 1	0.56	0.51	0.54	412201
2	Class 2	0.34	0.03	0.06	46475
3	Class 3	0.57	0.06	0.11	20564
4	Class 4	0.00	0.00	0.00	3775
5	Class 5	0.00	0.00	0.00	1942
6	Class 6	0.00	0.00	0.00	1386
7	Class 7	0.00	0.00	0.00	225
8	Class 8	0.00	0.00	0.00	14
9	Class 9	0.00	0.00	0.00	8

Key Observations:

1. **Performance Improvement**:
   - Classes `0` and `1` show notable improvements in precision, recall, and F1-score, contributing significantly to overall accuracy.
   - Accuracy improved by 6% compared to previous results, but rare classes continue to have very low predictive performance.
2. **Challenges with Minority Classes**:
   - Rare classes (e.g., `4`, `5`, `6`, `7`, `8`, and `9`) still have **precision**, **recall**, and **F1-scores** close to `0.00`.
   - This is likely due to class imbalance, where the model lacks sufficient examples to learn meaningful patterns for these classes.

0.8.4 Current Problem

- The current training set has increased in size to **50,000 hands**, which improved accuracy but was insufficient to address class imbalance.
- Minority classes are still not well represented, leading to poor performance for those classes in terms of recall and F1-score.

0.8.5 Plan: Apply SMOTE

- **SMOTE (Synthetic Minority Over-sampling Technique)** will be used to generate synthetic data points for the minority classes.
- This approach aims to balance the dataset, allowing the model to learn from more instances of the underrepresented classes.

0.8.6 Advantages of Using SMOTE

1. **Improved Representation of Minority Classes**:
   - SMOTE will generate synthetic examples, increasing the representation of minority classes such as `4`, `5`, `6`, etc., which will help the model generalise better for these categories.
2. **Better Performance Metrics**:
   - Balancing the dataset is expected to improve precision, recall, and F1-score for the underrepresented classes, leading to a more effective and generalised model.
3. **Enhanced Model Generalisation**:
   - By reducing class imbalance, the model will avoid biases towards majority classes, resulting in improved accuracy and performance across all classes.

[9]:
```
from imblearn.over_sampling import SMOTE  # Import SMOTE for generating␣
 ↪synthetic samples to balance class distribution in the training data

training_size = 50000  # Specify the desired size of the training set

# Calculate the total number of samples in the combined dataset
total_samples = combined_features.shape[0]  # Get the total number of samples

# Calculate the ratio of training samples to total samples
train_ratio = training_size / total_samples  # Ratio of samples to use for␣
 ↪training

# Split the combined dataset into training and test sets based on the training␣
 ↪ratio
X_train, X_test, y_train, y_test = train_test_split(
    combined_features, combined_target, test_size=(1 - train_ratio),␣
 ↪random_state=42
)

# Apply SMOTE to balance the training set by oversampling minority classes
smote = SMOTE(random_state=42,  k_neighbors=2)  # Create an instance of SMOTE␣
 ↪with a random seed for reproducibility and Reduce k_neighbors to avoid the␣
 ↪error
X_train, y_train = smote.fit_resample(X_train, y_train)  # Apply SMOTE to␣
 ↪generate synthetic samples for the training set

# Print the sizes of the balanced training set and the test set
print(f"Training Set Size after SMOTE: {X_train.shape[0]}")  # Size of the␣
 ↪training set after applying SMOTE
print(f"Test Set Size: {X_test.shape[0]}")  # Size of the test set
```

```
Training Set Size after SMOTE: 227538
Test Set Size: 975010
```

0.8.7 Observations:

- **Training Set Size after SMOTE**: Increased from **50,000** to **227,538** samples.
- **Test Set Size**: Remains at **975,010** samples.
- **Significant Increase**: Approximately **177,538 synthetic samples** were generated by SMOTE.

0.8.8 Interpretation:

- **Training Set Growth**: The substantial increase in training size indicates severe class imbalance in the original data, with SMOTE having to generate many synthetic samples.
- **Overfitting Risk**: The larger training set could cause the model to overfit to synthetic data rather than learning general patterns.

- **Training-Test Imbalance**: The imbalance between training (227,538 samples) and test set sizes (975,010 samples) might affect the model's ability to generalize effectively.

0.8.9 Conclusion:

- **Adjust SMOTE Strategy**: Consider reducing the number of synthetic samples by using a lower `sampling_strategy` or combining SMOTE with undersampling.
- **Model Complexity**: Control model complexity (e.g., limit tree depth) to avoid overfitting on the expanded training set.
- **Training Time**: Reducing synthetic samples can also help decrease the training time, making the model more practical to use.

[ ]:
```
# Train the Decision Tree Classifier on the SMOTE-balanced training set
model = DecisionTreeClassifier(random_state=42)  # Initialize the decision tree␣
 ↪model
model.fit(X_train, y_train)  # Train the model on the balanced training set

# Make predictions on the test set
y_pred = model.predict(X_test)  # Predict the target variable for the test set

# Evaluate the model using accuracy and classification report
print("Accuracy on Test Data:", accuracy_score(y_test, y_pred))  # Print the␣
 ↪accuracy on the test set
print("\nClassification Report:\n", classification_report(y_test, y_pred))  #␣
 ↪Print the classification report for the test set
```

```
Accuracy on Test Data: 0.4896206192756997

C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))


Classification Report:
              precision    recall  f1-score   support

           0       0.62      0.57      0.59    488420
           1       0.48      0.46      0.47    412201
           2       0.11      0.20      0.14     46475
           3       0.08      0.15      0.11     20564
           4       0.05      0.09      0.06      3775
           5       0.07      0.16      0.10      1942
           6       0.02      0.05      0.03      1386
           7       0.03      0.02      0.02       225
           8       0.02      0.07      0.03        14
           9       0.00      0.00      0.00         8
```

```
    accuracy                           0.49    975010
   macro avg       0.15      0.18      0.15    975010
weighted avg       0.52      0.49      0.50    975010
```

```
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
C:\Users\ronan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
The Kernel crashed while executing code in the current cell or a previous cell.

Please review the code in the cell(s) to identify a possible cause of the␣
↪failure.

Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info.

View Jupyter <a href='command:jupyter.viewOutput'>log</a> for further details.
```

0.8.10 SMOTE Performance Report

- **Training Set Size after SMOTE**: 227,538
- **Test Set Size**: 975,010

0.8.11 Accuracy

- **Accuracy on Test Data**: 48.96%

0.8.12 Classification Report

The updated classification report presents the performance of the decision tree classifier for each class after applying SMOTE:

Class	Category	Precision	Recall	F1-Score	Support
0	Class 0	0.62	0.57	0.59	488420
1	Class 1	0.48	0.46	0.47	412201
2	Class 2	0.11	0.20	0.14	46475

Class	Category	Precision	Recall	F1-Score	Support
3	Class 3	0.08	0.15	0.11	20564
4	Class 4	0.05	0.09	0.06	3775
5	Class 5	0.07	0.16	0.10	1942
6	Class 6	0.02	0.05	0.03	1386
7	Class 7	0.03	0.02	0.02	225
8	Class 8	0.02	0.07	0.03	14
9	Class 9	0.00	0.00	0.00	8

Key Observations:

1. **Improved Representation**
   - SMOTE increased the representation of minority classes, providing more balanced data for training.
   - However, despite this improvement, minority classes (e.g., `4`, `5`, `6`, `7`, `8`, and `9`) still have very low precision, recall, and F1-scores, indicating continued challenges in predictive performance.
2. **Performance Comparison**
   - Compared to the original training set of 50,000 hands, applying SMOTE actually led to a decrease in overall accuracy from 60% to 48.96%.
   - This drop in accuracy indicates that while SMOTE improved class balance, it may have introduced noise or overfitting, negatively impacting the model's ability to generalize.
   - More work is needed to improve the classifier's ability to effectively utilize the synthetic data generated by SMOTE for better generalization.

0.9 Data Sources

- **UCI Machine Learning Repository - Poker Hand Dataset**:
  - This dataset contains **1,000,000 instances** of poker hands, each represented by ten features:
    - **Features**: Suit (`S1-S5`) and rank (`R1-R5`) of five playing cards.
    - **Target Variable (`Class`)**: Indicates the type of poker hand, ranging from `0` (Nothing) to `9` (Royal Flush).
  - The dataset is split into a **training dataset** with 25,000 hands and a **test dataset** with 1,000,000 hands.

0.10 Pre-Processing

- **Data Splitting**:
  - Merged the original training and test datasets to create a unified dataset.
  - Split the combined data into new training and test sets with varying training sizes (**50,000**, **75,000**, and **100,000 hands**) using Scikit-learn's `train_test_split`.

  - Stratified sampling ensured consistent class distribution across training and test sets.

- **Feature Selection**:
  - Selected the ten features (`S1`, `R1`, …, `S5`, `R5`) representing the suits and ranks of each card.
  - No additional feature engineering was performed.

- **Handling Imbalanced Data**:
  - **SMOTE (Synthetic Minority Oversampling Technique)**:
    * Applied SMOTE to the 50,000-hand training set to address class imbalance.
    * Balanced the training data by generating synthetic samples for rare classes.
  - **Class Weighting**:
    * Considered adjusting class weights during model training to improve the recall of minority classes.

0.11 Data Understanding/Visualisation

- **Class Distribution Analysis**:
  - **Highly Imbalanced Classes**:
    * Most hands belong to class `0` (Nothing) and class `1` (One Pair).
    * Rare classes like `8` (Straight Flush) and `9` (Royal Flush) are significantly underrepresented.
  - **Visualization**:
    * Created bar charts to visualize class distribution and confirm the severity of the imbalance.

- **Impact of Imbalance**:
  - Imbalanced data leads to biased models that predict majority classes accurately but fail for rare classes.
  - Addressing this imbalance was crucial to improve the model's ability to generalize.

0.12 Algorithms

- **Decision Tree Classifier**:
  - Implemented using Scikit-learn's `DecisionTreeClassifier`.
  - **Cost Complexity Pruning**:
    * Controlled overfitting by simplifying the tree using the `ccp_alpha` parameter.
    * Evaluated the effect of different pruning strengths (`ccp_alpha`) on training and test accuracy.

- **Model Training and Evaluation**:
  - Trained models on datasets with varying training sizes (**50k**, **75k**, **100k**) to analyze scalability and performance.
  - Applied SMOTE to the 50,000-hand training set to address class imbalance.

- **Alternative Approaches**:
  - **Class Weighting**:
    - Adjusted class weights in the decision tree model to give more importance to rare classes.
  - **Explored Random Forests**:
    - Considered Random Forests as a potential alternative due to their ability to handle imbalanced datasets better.

0.13 Results

- **Initial Model Performance (Without Pruning)**:
  - The unpruned decision tree achieved **perfect accuracy** on training data but performed poorly on test data, indicating overfitting.
- **Pruning Analysis**:
  - **Optimal `ccp_alpha`**:
    - Identified an optimal pruning strength where test accuracy peaked at ~**54%**.
  - **Accuracy vs. Pruning Strength**:
    - Excessive pruning led to underfitting, causing both training and test accuracies to decline.
- **Increasing Training Set Size**:
  - Increasing the training set size to **50,000 hands** improved test accuracy to approximately **60%**.
  - Further increases to **75,000** and **100,000 hands** showed diminishing returns.
- **SMOTE Results**:
  - **Impact on Accuracy**:
    - SMOTE reduced overall test accuracy to ~**49%**, highlighting challenges in generalizing to test data after oversampling.
  - **Minority Class Performance**:
    - While SMOTE balanced the training data, the model still struggled to predict rare classes effectively.
- **Classification Report Highlights**:
  - **Majority Classes** (`0`, `1`):
    - Achieved better precision and recall due to their prevalence in the dataset.
  - **Minority Classes** (`8`, `9`):
    - Precision and recall remained close to zero, reflecting the model's difficulty in learning these rare classes.

0.14 Online Resources & Sources

- **UCI Machine Learning Repository - Poker Hand Dataset**:

  - Provided the dataset for this project.
- **Scikit-learn Documentation**:
  - Assisted in implementing decision trees, pruning, and evaluation metrics.
- **Machine Learning Mastery - SMOTE**:
  - Helped apply SMOTE to handle class imbalance.
- **Matplotlib Documentation**:
  - Used for creating visualizations of class distributions and model performance.

0.15 Tools & Technologies Used

- **Python Libraries**:
  - `Pandas` and `NumPy`: Data preprocessing and manipulation.
  - `Scikit-learn`: Implemented decision trees, pruning, and evaluation metrics.
  - `Imbalanced-Learn`: Applied SMOTE for oversampling minority classes.
- **Visualization**:
  - `Matplotlib`: Plotted class distributions and pruning results.
- **Development Environment**:
  - **Jupyter Notebook**: Used for interactive data analysis and model training.

0.16 Challenges Faced

1. **Class Imbalance**:
   - The dataset was heavily imbalanced, with most samples in classes `0` and `1`.
   - Techniques like SMOTE and class weighting improved balance but did not fully resolve performance issues for rare classes.
2. **Computational Overhead**:
   - Increasing the training set size to 100,000 hands and applying pruning led to longer training times.
   - Pruning became computationally expensive for larger datasets.
3. **SMOTE Trade-Offs**:
   - While SMOTE balanced the training data, it introduced synthetic samples that reduced overall test accuracy.
   - The model failed to generalize well to the test set after SMOTE application.
4. **Pruning Complexity**:
   - Finding the optimal `ccp_alpha` required extensive experimentation and visualizations.
   - Over-pruned models underfit the data, losing predictive power.
5. **Rare Class Predictions**:
   - Despite balancing efforts, the model struggled to predict rare classes like `8` (Straight Flush) and `9` (Royal Flush).
   - These classes consistently had near-zero precision and recall.