```
!wget https://zenodo.org/records/8357398
```

```
--2025-05-26 11:04:11--  https://zenodo.org/records/8357398
Resolving zenodo.org (zenodo.org)... 188.185.43.25, 188.185.48.194, 188.185.45.92, ...
Connecting to zenodo.org (zenodo.org)|188.185.43.25|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 93848 (92K) [text/html]
Saving to: ‘8357398.1’

8357398.1           100%[===================>]  91.65K   196KB/s    in 0.5s

2025-05-26 11:04:12 (196 KB/s) - ‘8357398.1’ saved [93848/93848]
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
```

```
# Install the cyvcf2 library
!pip install cyvcf2
# Now import the necessary libraries
from cyvcf2 import VCF
import numpy as np
import pandas as pd

# Use the path to your VCF file in Google Drive
vcf = VCF('/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf')
```

```
Requirement already satisfied: cyvcf2 in /usr/local/lib/python3.11/dist-packages (0.31.1)
Requirement already satisfied: numpy>=1.16.0 in /usr/local/lib/python3.11/dist-packages (from cyvcf2) (2.0.2)
Requirement already satisfied: coloredlogs in /usr/local/lib/python3.11/dist-packages (from cyvcf2) (15.0.1)
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-packages (from cyvcf2) (8.2.0)
Requirement already satisfied: humanfriendly>=9.1 in /usr/local/lib/python3.11/dist-packages (from coloredlogs->cyvcf2) (10.0)
```

```
gt_df = pd.DataFrame()
gt_df.head()
```

```
from cyvcf2 import VCF

vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Print the number of samples and the first few sample names
print(f"Number of samples: {len(vcf.samples)}")
print(f"First 5 sample names: {vcf.samples[:5]}")
```

```
Number of samples: 153
First 5 sample names: ['aethiopicum1', 'aethiopicum2', 'aethiopicum3', 'aethiopicum4', 'macrocarpon1']
```

```
# Print information for the first 5 variants
print("Information for the first 5 variants:")
for i, variant in enumerate(vcf):
  if i >= 5:
    break
print(f"Variant {i+1}: Chromosome={variant.CHROM}, Position={variant.POS}, Ref={variant.REF}, Alts={variant.ALT}")

# Close the VCF object when done
vcf.close()
```

```
Information for the first 5 variants:
Variant 6: Chromosome=0, Position=259625, Ref=A, Alts=['G']
```

```
!head -n 50 "/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf"
```

```
##fileformat=VCFv4.0
##Tassel=<ID=GenotypeTable,Version=5,Description="Reference allele is not known. The major allele was used as reference allele">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=AD,Number=.,Type=Integer,Description="Allelic depths for the reference and alternate alleles in the order listed">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth (only filtered reads used for calling)">
##FORMAT=<ID=GQ,Number=1,Type=Float,Description="Genotype Quality">
##FORMAT=<ID=PL,Number=.,Type=Float,Description="Normalized, Phred-scaled likelihoods for AA,AB,BB genotypes where A=ref and B=a
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  aethiopicum1    aethiopicum2    aethiopicum3    aethiop
0       8459    50:139:-        G       A       .       PASS    AF=0.102;NS=254;DP=148  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       8569    50:29:- A       C       .       PASS    AF=0.054;NS=257;DP=149  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36     0/0:1,0
0       10795   75:29:+ A       C       .       PASS    AF=0.032;NS=251;DP=184  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36     0/0:1,0
0       10881   75:115:+        T       C       .       PASS    AF=0.371;NS=251;DP=183  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       259553  341:88:+        A       T       .       PASS    AF=0.344;NS=262;DP=170  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       259625  341:160:+       A       G       .       PASS    AF=0.255;NS=259;DP=168  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       1553460 1361:40:+       G       A       .       PASS    AF=0.289;NS=246;DP=175  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       1947362 1683:9:+        C       T       .       PASS    AF=0.491;NS=224;DP=153  GT:AD:DP:GQ:PL  1/1:0,1:1:66:36,3,0
0       1947420 1683:67:+       C       A       .       PASS    AF=0.124;NS=225;DP=153  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2031644 1872:13:+       C       T       .       PASS    AF=0.204;NS=245;DP=194  GT:AD:DP:GQ:PL  1/1:0,1:1:66:36,3,0
0       2034430 1902:10:-       T       C       .       PASS    AF=0.037;NS=243;DP=214  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2420258 2296:16:+       A       G       .       PASS    AF=0.154;NS=234;DP=170  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2420393 2296:151:+      C       T       .       PASS    AF=0.157;NS=235;DP=169  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2438536 2362:199:-      G       A       .       PASS    AF=0.457;NS=242;DP=154  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2438691 2362:44:-       C       T       .       PASS    AF=0.053;NS=244;DP=210  GT:AD:DP:GQ:PL  0/0:2,0:2:79:0,6,72
0       2438840 2363:83:+       G       T       .       PASS    AF=0.462;NS=263;DP=158  GT:AD:DP:GQ:PL  0/0:2,0:2:79:0,6,72
0       2438993 2363:236:+      G       A       .       PASS    AF=0.355;NS=262;DP=156  GT:AD:DP:GQ:PL  0/0:2,0:2:79:0,6,72
0       2510178 2500:14:+       T       A       .       PASS    AF=0.112;NS=241;DP=168  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2510248 2500:84:+       A       G       .       PASS    AF=0.033;NS=239;DP=167  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2559723 2625:27:+       T       A       .       PASS    AF=0.062;NS=275;DP=210  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       2559867 2625:171:+      A       G       .       PASS    AF=0.144;NS=275;DP=209  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       3235934 3362:21:+       A       C       .       PASS    AF=0.144;NS=236;DP=168  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       3236004 3362:91:+       T       C       .       PASS    AF=0.485;NS=235;DP=165  GT:AD:DP:GQ:PL  1/1:0,1:1:66:36,3,0
0       3236077 3362:164:+      C       T       .       PASS    AF=0.097;NS=236;DP=167  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       3393132 3572:141:+      A       G       .       PASS    AF=0.406;NS=229;DP=159  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       3813647 4128:23:+       T       C       .       PASS    AF=0.086;NS=243;DP=147  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       3813832 4132:46:+       G       C       .       PASS    AF=0.202;NS=253;DP=157  GT:AD:DP:GQ:PL  1/1:0,2:2:79:72,6,0
0       3813973 4135:18:+       C       T       .       PASS    AF=0.035;NS=227;DP=137  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       3814093 4143:94:-       T       C       .       PASS    AF=0.184;NS=239;DP=167  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       4689574 5174:8:+        A       T       .       PASS    AF=0.257;NS=239;DP=247  GT:AD:DP:GQ:PL  0/0:2,0:2:79:0,6,72
```

```
0       4689655 5174:89:+       C       A       .       PASS    AF=0.240;NS=233;DP=237  GT:AD:DP:GQ:PL  0/0:2,0:2:79:0,6,72
0       4690095 5183:22:+       A       G       .       PASS    AF=0.147;NS=238;DP=160  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       4690146 5183:73:+       C       G       .       PASS    AF=0.211;NS=227;DP=157  GT:AD:DP:GQ:PL  1/1:0,1:1:66:36,3,0
0       4741482 5223:11:+       T       C       .       PASS    AF=0.455;NS=244;DP=169  GT:AD:DP:GQ:PL  0/0:2,0:2:79:0,6,72
0       4792027 5423:75:+       C       A       .       PASS    AF=0.340;NS=247;DP=164  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       4792094 5423:142:+      T       C       .       PASS    AF=0.077;NS=246;DP=164  GT:AD:DP:GQ:PL  0/0:1,0:1:66:0,3,36
0       5252678 5799:166:-      T       C       .       PASS    AF=0.035;NS=270;DP=202  GT:AD:DP:GQ:PL  0/1:0,2:2:79:72,6,0
0       5252733 5793:77:+       T       A       .       PASS    AF=0.034;NS=277;DP=332  GT:AD:DP:GQ:PL  0/0:3,0:3:88:0,9,108
0       5637721 6247:23:+       T       C       .       PASS    AF=0.094;NS=235;DP=358  GT:AD:DP:GQ:PL  0/0:3,0:3:88:0,9,108
```

Start coding or generate with AI.

```
# Function to parse genotypes and convert to numerical representation
def gt_to_numeric(gt):
    """
    Converts a VCF genotype string to a numerical representation.

    Args:
        gt (str): The genotype string (e.g., '0/0', '0/1', '1/1', './.').

    Returns:
        int or np.nan: 0 for homozygous reference, 1 for heterozygous,
                       2 for homozygous alternate, or np.nan for missing.
    """
    if gt in ('./.', '.|.'):
        return np.nan  # Missing genotype
    elif gt in ('0/0', '0|0'):
        return 0  # Homozygous reference
    elif gt in ('0/1', '1/0', '0|1', '1|0'):
        return 1  # Heterozygous
    elif gt in ('1/1', '1|1'):
        return 2  # Homozygous alternate
    else:
        # Handle other possible genotype formats if needed, or raise an error
        return np.nan # Default to missing for unhandled formats
```

```
# Store all variants in a dataframe
# Make DataFrame showing the chromosome_position as row index,
# the samples as column names, and a numerical value of the GT as content.

# Re-open the VCF file as the previous loop would have closed it
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Initialize lists to store data for the DataFrame
variant_positions = []
sample_genotypes = []

# Iterate through each variant in the VCF file
for variant in vcf:
    # Create a unique identifier for the variant using Chromosome and Position
    variant_id = f"{variant.CHROM}_{variant.POS}"
    variant_positions.append(variant_id)

    # Get genotypes for all samples for the current variant
    # variant.gt_types provides a numpy array of integers representing genotypes
    # 0: Homozygous reference, 1: Heterozygous, 2: Homozygous alternate, 3: Missing
    # We'll convert these to our desired numerical format using gt_to_numeric
    genotype_row = [gt_to_numeric(variant.genotypes[i][0]) for i in range(len(vcf.samples))]
    sample_genotypes.append(genotype_row)

# Get the list of sample names from the VCF object
samples = vcf.samples

# Close the VCF object
vcf.close()

# Create the DataFrame
# The rows will be variant positions, columns will be sample names, and values will be numerical genotypes
gt_df = pd.DataFrame(sample_genotypes, index=variant_positions, columns=samples)

# Display the first few rows of the created DataFrame
display(gt_df.head())
```

	aethiopicum1	aethiopicum2	aethiopicum3	aethiopicum4	macrocarpon1	macrocarpon2	macrocarpon3	dasyphyllum1	dasyph
0_8459	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
0_8569	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
0_10795	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
0_10881	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
0_259553	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

5 rows × 153 columns

```
# Check for the sum of null values in the filtered DataFrame

# Count the number of null values in each column
null_counts_per_column = filtered_gt_df.isnull().sum()

# Calculate the total sum of null values across the entire DataFrame
total_null_count = null_counts_per_column.sum()

# Print the results
print("Number of null values per sample (column):")
print(null_counts_per_column)

print(f"\nTotal number of null values in the filtered DataFrame: {total_null_count}")
```

```
Number of null values per sample (column):
aethiopicum1    0
aethiopicum2    0
aethiopicum3    0
aethiopicum4    0
macrocarpon1    0
               ..
incanum15       0
incanum16       0
incanum17       0
```

```
incanum18       0
incanum19       0
Length: 153, dtype: int64

Total number of null values in the filtered DataFrame: 0
```

```
# map samples to groups
from collections import defaultdict

# Extract group names by trimming digits from the end
def extract_group(sample):
    """
    Extracts a group name from a sample string by removing trailing digits.

    Args:
        sample (str): The sample name string (e.g., 'SampleA1', 'SampleB05').

    Returns:
        str: The extracted group name (e.g., 'SampleA', 'SampleB').
    """
    # Find the index of the last non-digit character from the right
    i = len(sample) - 1
    while i >= 0 and sample[i].isdigit():
        i -= 1
    # Return the substring up to (and including) the last non-digit character
    return sample[:i+1]

# Create a defaultdict to store sample groups
sample_groups = defaultdict(list)

# Get the sample names from the previously created gt_df DataFrame
samples = gt_df.columns

# Iterate through each sample and extract its group
for sample in samples:
    group = extract_group(sample)
    sample_groups[group].append(sample)
```

```
# Display the sample groups (optional)
print("Sample Groups:")
for group, sample_list in sample_groups.items():
    print(f"{group}: {sample_list}")
```

```
Sample Groups:
aethiopicum: ['aethiopicum1', 'aethiopicum2', 'aethiopicum3', 'aethiopicum4']
macrocarpon: ['macrocarpon1', 'macrocarpon2', 'macrocarpon3', 'macrocarpon4', 'macrocarpon5', 'macrocarpon6', 'macrocarpon7', 'm
dasyphyllum: ['dasyphyllum1', 'dasyphyllum2', 'dasyphyllum3', 'dasyphyllum4', 'dasyphyllum5', 'dasyphyllum6', 'dasyphyllum7', 'd
anomalum: ['anomalum2', 'anomalum3', 'anomalum4', 'anomalum5', 'anomalum6', 'anomalum7', 'anomalum8', 'anomalum9', 'anomalum10',
anguivi: ['anguivi2', 'anguivi3', 'anguivi4', 'anguivi5', 'anguivi6', 'anguivi7', 'anguivi8', 'anguivi10', 'anguivi11', 'anguivi
cerasiferum: ['cerasiferum1', 'cerasiferum2', 'cerasiferum3', 'cerasiferum4', 'cerasiferum5', 'cerasiferum6', 'cerasiferum7', 'd
incanum: ['incanum1', 'incanum2', 'incanum3', 'incanum4', 'incanum5', 'incanum6', 'incanum7', 'incanum8', 'incanum9', 'incanum10
coagulans: ['coagulans1', 'coagulans2', 'coagulans3', 'coagulans5', 'coagulans6', 'coagulans7']
aculeastrum: ['aculeastrum1']
aculeatissimum: ['aculeatissimum1', 'aculeatissimum2']
arundo: ['arundo1']
campylacanthum: ['campylacanthum1', 'campylacanthum2', 'campylacanthum3', 'campylacanthum4', 'campylacanthum5', 'campylacanthum6
sp: ['sp1', 'sp2', 'sp3', 'sp4']
dasyanthum: ['dasyanthum1', 'dasyanthum2']
mauense: ['mauense1', 'mauense2']
nigriviolaceum: ['nigriviolaceum1']
phoxocarpum: ['phoxocarpum1']
setaceum: ['setaceum1', 'setaceum2']
```

```
# filter the data based on QUAL or FILTER field. State your filtering criteria (pick any values)

# Filtering criteria:
# - QUAL score must be greater than 30
# - FILTER field must be "PASS"

# Re-open the VCF file for filtering
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Initialize lists to store data for the filtered DataFrame
```

```
filtered_variant_positions = []
filtered_sample_genotypes = []

# Iterate through each variant and apply filtering criteria
for variant in vcf:
    # Apply filtering based on QUAL and FILTER fields
    if variant.QUAL is not None and variant.QUAL > 30 and "PASS" in variant.FILTER:
        # If the variant passes the filter, store its information
        variant_id = f"{variant.CHROM}_{variant.POS}"
        filtered_variant_positions.append(variant_id)

        # Get genotypes for all samples for the current variant
        genotype_row = [gt_to_numeric(variant.genotypes[i][0]) for i in range(len(vcf.samples))]
        filtered_sample_genotypes.append(genotype_row)

# Get the list of sample names from the VCF object (assuming the same samples as before)
samples = vcf.samples

# Close the VCF object
vcf.close()

# Create the filtered DataFrame
# The rows will be filtered variant positions, columns will be sample names, and values will be numerical genotypes
filtered_gt_df = pd.DataFrame(filtered_sample_genotypes, index=filtered_variant_positions, columns=samples)

# Display the first few rows of the filtered DataFrame
display(filtered_gt_df.head())

#Would you proceed based on the output?

# Proceed if the filtered DataFrame is not empty, indicating that some variants met the criteria.
# If the filtered DataFrame is empty, the chosen criteria were too strict or there are no variants meeting them.
if not filtered_gt_df.empty:
    print("\nFiltered DataFrame created successfully. Proceeding with analysis.")
else:
    print("\nFiltered DataFrame is empty. No variants met the specified filtering criteria. Consider adjusting the criteria.")
```

aethiopicum1	aethiopicum2	aethiopicum3	aethiopicum4	macrocarpon1	macrocarpon2	macrocarpon3	dasyphyllum1	dasyphyllum2

0 rows × 153 columns

```
Filtered DataFrame is empty. No variants met the specified filtering criteria. Consider adjusting the criteria.
```

Start coding or generate with AI.

```
# filter the data based on QUAL or FILTER field. State your filtering criteria (pick any values)

# Filtering criteria (Adjusted for leniency):
# Option 1: Lower the QUAL threshold
# Option 2: Allow variants that PASS OR meet a lower QUAL threshold (if FILTER != PASS)
# Option 3: Focus on FILTER field only (e.g., include variants that are not filtered out by "LowQual")
# Option 4: Remove the QUAL filtering entirely and only filter by FILTER

# Let's try Option 1: Lower the QUAL threshold and keep the "PASS" filter

# Re-open the VCF file for filtering
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Initialize lists to store data for the filtered DataFrame
filtered_variant_positions = []
filtered_sample_genotypes = []

# Iterate through each variant and apply filtering criteria
for variant in vcf:
    # Apply filtering based on QUAL and FILTER fields (Adjusted)
    # Option 1: Lower the QUAL threshold (e.g., to 20 or 10)
    # Change 30 to a lower value like 20: variant.QUAL > 20
    # Or even lower, like 10: variant.QUAL > 10
    # Or even just require QUAL is not None and FILTER is PASS: variant.QUAL is not None and "PASS" in variant.FILTER
    # Let's use QUAL > 20 as an example
    if variant.QUAL is not None and variant.QUAL > 5 and "PASS" in variant.FILTER:
```

```
        # You could also consider other options:
        # Option 2: Allow variants that PASS OR meet a lower QUAL threshold (if FILTER != PASS)
        # if ("PASS" in variant.FILTER) or (variant.QUAL is not None and variant.QUAL > 10):

        # Option 3: Focus on FILTER field only (e.g., include variants that are not filtered out by "LowQual")
        # if "LowQual" not in variant.FILTER:

        # Option 4: Remove the QUAL filtering entirely and only filter by FILTER
        # if "PASS" in variant.FILTER:

            # If the variant passes the filter, store its information
            variant_id = f"{variant.CHROM}_{variant.POS}"
            filtered_variant_positions.append(variant_id)

            # Get genotypes for all samples for the current variant
            genotype_row = [gt_to_numeric(variant.genotypes[i][0]) for i in range(len(vcf.samples))]
            filtered_sample_genotypes.append(genotype_row)

    # Get the list of sample names from the VCF object (assuming the same samples as before)
    samples = vcf.samples

    # Close the VCF object
    vcf.close()

    # Create the filtered DataFrame
    # The rows will be filtered variant positions, columns will be sample names, and values will be numerical genotypes
    filtered_gt_df = pd.DataFrame(filtered_sample_genotypes, index=filtered_variant_positions, columns=samples)

    # Display the first few rows of the filtered DataFrame
    display(filtered_gt_df.head())

    #Would you proceed based on the output?

    # Proceed if the filtered DataFrame is not empty, indicating that some variants met the criteria.
    # If the filtered DataFrame is empty, the chosen criteria were too strict or there are no variants meeting them.
    if not filtered_gt_df.empty:
```

```
    print("\nFiltered DataFrame created successfully. Proceeding with analysis.")
else:
    print("\nFiltered DataFrame is empty. No variants met the specified filtering criteria. Consider adjusting the criteria.")
```

aethiopicum1	aethiopicum2	aethiopicum3	aethiopicum4	macrocarpon1	macrocarpon2	macrocarpon3	dasyphyllum1	dasyphyllum2

0 rows × 153 columns

```
Filtered DataFrame is empty. No variants met the specified filtering criteria. Consider adjusting the criteria.
```

Start coding or generate with AI.

```
# filter the data based on QUAL or FILTER field. State your filtering criteria (pick any values)

# Filtering criteria (Removing QUAL filter):
# - FILTER field must be "PASS"

# Re-open the VCF file for filtering
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Initialize lists to store data for the filtered DataFrame
filtered_variant_positions = []
filtered_sample_genotypes = []

# Iterate through each variant and apply filtering criteria
for variant in vcf:
    # Apply filtering based only on the FILTER field
    # Add a check to ensure variant.FILTER is not None before checking for "PASS"
    if variant.FILTER is not None and "PASS" in variant.FILTER:

        # If the variant passes the filter, store its information
        variant_id = f"{variant.CHROM}_{variant.POS}"
        filtered_variant_positions.append(variant_id)
```

```
        # Get genotypes for all samples for the current variant
        genotype_row = [gt_to_numeric(variant.genotypes[i][0]) for i in range(len(vcf.samples))]
        filtered_sample_genotypes.append(genotype_row)

# Get the list of sample names from the VCF object (assuming the same samples as before)
samples = vcf.samples

# Close the VCF object
vcf.close()

# Create the filtered DataFrame
# The rows will be filtered variant positions, columns will be sample names, and values will be numerical genotypes
filtered_gt_df = pd.DataFrame(filtered_sample_genotypes, index=filtered_variant_positions, columns=samples)

# Display the first few rows of the filtered DataFrame
display(filtered_gt_df.head())

#Would you proceed based on the output?

# Proceed if the filtered DataFrame is not empty, indicating that some variants met the criteria.
# If the filtered DataFrame is empty, the chosen criteria were too strict or there are no variants meeting them.
if not filtered_gt_df.empty:
    print("\nFiltered DataFrame created successfully. Proceeding with analysis.")
else:
    print("\nFiltered DataFrame is empty. No variants met the specified filtering criteria. Consider adjusting the criteria.")
```

aethiopicum1	aethiopicum2	aethiopicum3	aethiopicum4	macrocarpon1	macrocarpon2	macrocarpon3	dasyphyllum1	dasyphyllum2

0 rows × 153 columns

```
Filtered DataFrame is empty. No variants met the specified filtering criteria. Consider adjusting the criteria.
```

Start coding or generate with AI.

```
# Re-open the VCF file to access the header
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Access and display specific metadata lines
print("--- Key VCF Header Metadata ---")

# Get and print the file format version from the raw header
for line in vcf.raw_header.splitlines():
    if line.startswith('##fileformat'):
        print(f"File Format: {line.split('=')[1]}")
        break

# Get and print source information (often contains software used)
# Since get_source() might not be available, you can parse it from raw_header if needed
source_info = [line for line in vcf.raw_header.splitlines() if line.startswith('##source')]
if source_info:
    print(f"Source: {source_info[0].split('=')[1]}")

# Get and print reference genome information
# Since get_reference() might not be available, you can parse it from raw_header if needed
reference_info = [line for line in vcf.raw_header.splitlines() if line.startswith('##reference')]
if reference_info:
    print(f"Reference: {reference_info[0].split('=')[1]}")

# Get and print assembly information
# Since get_assembly() might not be available, you can parse it from raw_header if needed
assembly_info = [line for line in vcf.raw_header.splitlines() if '##assembly' in line]
if assembly_info:
    print(f"Assembly: {assembly_info[0].split('=')[1]}")

# Rest of your code remains the same
```

```
--- Key VCF Header Metadata ---
File Format: VCFv4.0
```

Start coding or generate with AI.

```
# Code to access and display VCF header metadata programmatically

# Re-open the VCF file to access the header
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

# Access the raw header string
print("--- Raw VCF Header ---")
print(vcf.raw_header)
print("----------------------")

# Access and display specific metadata lines
print("--- Key VCF Header Metadata ---")

# Get and print the file format version from the raw header
for line in vcf.raw_header.splitlines():
    if line.startswith('##fileformat'):
        print(f"File Format: {line.split('=')[1]}")
        break

# Get and print source information
source_info = [line for line in vcf.raw_header.splitlines() if line.startswith('##source')]
if source_info:
    print(f"Source: {source_info[0].split('=')[1]}")
else:
    print("No source information found in header.")

# Get and print reference genome information
reference_info = [line for line in vcf.raw_header.splitlines() if line.startswith('##reference')]
if reference_info:
    print(f"Reference: {reference_info[0].split('=')[1]}")
else:
    print("No reference information found in header.")

# Get and print assembly information
```

```
assembly_info = [line for line in vcf.raw_header.splitlines() if line.startswith('##assembly')]
if assembly_info:
    print(f"Assembly: {assembly_info[0].split('=')[1]}")
else:
    print("No assembly information found in header.")
```

```
# Access and print sample names
print("\n--- Sample Names from Header ---")
if vcf.samples:
    print(vcf.samples)
else:
    print("No sample names found in header.")

# You can also iterate through all header lines
# print("\n--- All Header Lines ---")
# Use raw_header to iterate through header lines
# for line in vcf.raw_header.splitlines():
#     print(line)

# Close the VCF object when done
vcf.close()
```

```
--- Raw VCF Header ---
##fileformat=VCFv4.0
##FILTER=<ID=PASS,Description="All filters passed">
##Tassel=<ID=GenotypeTable,Version=5,Description="Reference allele is not known. The major allele was used as reference allele">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=AD,Number=.,Type=Integer,Description="Allelic depths for the reference and alternate alleles in the order listed">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth (only filtered reads used for calling)">
##FORMAT=<ID=GQ,Number=1,Type=Float,Description="Genotype Quality">
```

```
##FORMAT=<ID=PL,Number=.,Type=Float,Description="Normalized, Phred-scaled likelihoods for AA,AB,BB genotypes where A=ref and B=a
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  aethiopicum1    aethiopicum2    aethiopicum3    aethiop

----------------------
--- Key VCF Header Metadata ---
File Format: VCFv4.0
No source information found in header.
No reference information found in header.
No assembly information found in header.

--- Sample Names from Header ---
['aethiopicum1', 'aethiopicum2', 'aethiopicum3', 'aethiopicum4', 'macrocarpon1', 'macrocarpon2', 'macrocarpon3', 'dasyphyllum1',
```

```
# Access and print FORMAT field definitions
print("\n--- FORMAT Field Definitions ---")
format_fields = vcf.formats
if format_fields:
    for fmt in format_fields:
        print(f"ID: {fmt.id}, Number: {fmt.number}, Type: {fmt.type}, Description: {fmt.description}")
else:
    print("No FORMAT field definitions found in header.")


print("\n--- FORMAT Field Definitions ---")
format_fields = list(reader.header.formats.values())
if format_fields:
    for fmt in format_fields:
        print(f"ID: {fmt.id}, Number: {fmt.number}, Type: {fmt.type}, Description: {fmt.description}")
else:
    print("No FORMAT field definitions found in header.")


# Access and print contig information
print("\n--- Contig Information ---")
contigs = vcf.contigs
```

```
if contigs:
    for contig in contigs:
        print(f"ID: {contig.id}, Length: {contig.length}")
else:
    print("No Contig information found in header.")
```

Start coding or generate with AI.

```
import pandas as pd
import numpy as np
from cyvcf2 import VCF
from collections import defaultdict
from IPython.display import display # Import display for cleaner output in notebooks

# Assuming gt_to_numeric function is defined in a previous cell
# If not, define it here:
def gt_to_numeric(gt):
    """
    Converts a VCF genotype string to a numerical representation.

    Args:
        gt (str): The genotype string (e.g., '0/0', '0/1', '1/1', './.').

    Returns:
        int or np.nan: 0 for homozygous reference, 1 for heterozygous,
                       2 for homozygous alternate, or np.nan for missing.
    """
    if gt in ('./.', '.|.'):
        return np.nan  # Missing genotype
    elif gt in ('0/0', '0|0'):
        return 0  # Homozygous reference
    elif gt in ('0/1', '1/0', '0|1', '1|0'):
        return 1  # Heterozygous
    elif gt in ('1/1', '1|1'):
        return 2  # Homozygous alternate
    else:
```

```
        # Handle other possible genotype formats if needed, or raise an error
        return np.nan # Default to missing for unhandled formats

# Specify the path to your VCF file
# Make sure the path is correct. It was '/content/Wild_African_eggplant_SNP_dataset.vcf.gz' in the prompt.
# If you are using the file mounted from Google Drive, use the path from the previous cells.
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
# vcf_path = '/content/Wild_African_eggplant_SNP_dataset.vcf.gz' # Use this if you downloaded it directly

# Open the VCF file
vcf = VCF(vcf_path)

samples = vcf.samples
num_samples = len(samples)

# Initialize counters for heterozygosity and missingness
# Count of heterozygous variants per sample
het_counts_per_sample = defaultdict(int)
# Count of missing variants per sample
missing_counts_per_sample = defaultdict(int)
# Total number of variants processed (for calculating sample rates)
total_variants_processed = 0

# Count of heterozygous genotypes per variant
het_counts_per_variant = []
# Count of missing genotypes per variant
missing_counts_per_variant = []
# List to store variant IDs
variant_ids = []


# Iterate through each variant in the VCF file
for variant in vcf:
    total_variants_processed += 1
    variant_id = f"{variant.CHROM}_{variant.POS}"
    variant_ids.append(variant_id)
```

```
        # Counters for the current variant
        variant_het_count = 0
        variant_missing_count = 0

        # Iterate through genotypes for each sample for the current variant
        for i, sample_name in enumerate(samples):
            # Access the raw genotype string (e.g., '0/1', './.')
            # variant.genotypes is a list of lists, where each inner list is [GT_integer, allele1, allele2, phasing]
            # We need to reconstruct the string representation to use gt_to_numeric accurately,
            # or alternatively, interpret the integer representation directly.
            # Let's use the integer representation from variant.gt_types for efficiency.
            # 0: Homozygous ref, 1: Heterozygous, 2: Homozygous alt, 3: Missing

            gt_int = variant.gt_types[i]

            if gt_int == 1: # Heterozygous (0/1 or 1/0)
                het_counts_per_sample[sample_name] += 1
                variant_het_count += 1
            elif gt_int == 3: # Missing (./. or .|.)
                missing_counts_per_sample[sample_name] += 1
                variant_missing_count += 1
            # Note: Homozygous ref (0) and Homozygous alt (2) don't increment these counters

        # Store variant-level counts
        het_counts_per_variant.append(variant_het_count)
        missing_counts_per_variant.append(variant_missing_count)


    # Close the VCF object
    vcf.close()

    # Calculate rates for samples
    # Total number of variants is total_variants_processed
    sample_het_rates = {sample: count / total_variants_processed for sample, count in het_counts_per_sample.items()}
    sample_missing_rates = {sample: count / total_variants_processed for sample, count in missing_counts_per_sample.items()}

    # Create sample_df
```

```
sample_data = {
    'Heterozygosity_Rate': pd.Series(sample_het_rates),
    'Missingness_Rate': pd.Series(sample_missing_rates)
}
sample_df = pd.DataFrame(sample_data)

print("Sample Rates (Heterozygosity and Missingness):")
display(sample_df.head()) # Use display for better formatting in notebooks

# Calculate rates for variants
# Total number of samples is num_samples
snp_het_rates = [count / num_samples for count in het_counts_per_variant]
snp_missing_rates = [count / num_samples for count in missing_counts_per_variant]

# Create snp_df
snp_data = {
    'Heterozygosity_Rate': snp_het_rates,
    'Missingness_Rate': snp_missing_rates
}
snp_df = pd.DataFrame(snp_data, index=variant_ids)

print("\nSNP Rates (Heterozygosity and Missingness):")
display(snp_df.head()) # Use display for better formatting in notebooks

# Optional: Display shapes of the dataframes
print(f"\nShape of sample_df: {sample_df.shape}")
print(f"Shape of snp_df: {snp_df.shape}")
```

```
Sample Rates (Heterozygosity and Missingness):
```

	Heterozygosity_Rate	Missingness_Rate
aculeastrum1	0.003103	0.239469
aculeatissimum1	0.002971	0.108940
aculeatissimum2	0.001915	0.097187
aethiopicum1	0.004292	0.190413
aethiopicum2	0.006140	0.194771

```
SNP Rates (Heterozygosity and Missingness):
```

	Heterozygosity_Rate	Missingness_Rate
0_8459	0.0	0.111111
0_8569	0.0	0.091503
0_10795	0.0	0.052288
0_10881	0.0	0.143791
0_259553	0.0	0.124183

```
Shape of sample_df: (153, 2)
Shape of snp_df: (15146, 2)
```

Start coding or generate with AI.

```
# Continue from the previous cell where sample_df and snp_df were created

import matplotlib.pyplot as plt
import seaborn as sns # Often used for better looking plots

# Set style for plots
sns.set_style("whitegrid")
```

```
print("--- Examining sample_df ---")

# 1. View the first few rows (already done, but good for completeness)
print("\nSample Rates Head:")
display(sample_df.head())

# 2. Get Summary Statistics for sample_df
print("\nSample Rates Summary Statistics:")
display(sample_df.describe())

# 3. Check for Missing Values in sample_df
print("\nMissing values per column in sample_df:")
print(sample_df.isnull().sum()) # Should likely be all zeros

# 4. Visualize Distributions for sample_df
print("\nVisualizing Distributions for Sample Rates:")
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1) # 1 row, 2 columns, 1st plot
sns.histplot(sample_df['Heterozygosity_Rate'], kde=True)
plt.title('Distribution of Sample Heterozygosity Rates')
plt.xlabel('Heterozygosity Rate')
plt.ylabel('Count')

plt.subplot(1, 2, 2) # 1 row, 2 columns, 2nd plot
sns.histplot(sample_df['Missingness_Rate'], kde=True, color='salmon')
plt.title('Distribution of Sample Missingness Rates')
plt.xlabel('Missingness Rate')
plt.ylabel('Count')

plt.tight_layout() # Adjust layout to prevent overlapping
plt.show()

# 5. Sort to find samples with highest/lowest rates
print("\nSamples with Highest Heterozygosity Rate:")
display(sample_df.sort_values(by='Heterozygosity_Rate', ascending=False).head())

print("\nSamples with Highest Missingness Rate:")
```

```
print("\nSamples with Highest Missingness Rate:")
display(sample_df.sort_values(by='Missingness_Rate', ascending=False).head())


print("\n--- Examining snp_df ---")

# 1. View the first few rows (already done, but good for completeness)
print("\nSNP Rates Head:")
display(snp_df.head())

# 2. Get Summary Statistics for snp_df
print("\nSNP Rates Summary Statistics:")
display(snp_df.describe())

# 3. Check for Missing Values in snp_df
print("\nMissing values per column in snp_df:")
print(snp_df.isnull().sum()) # Should likely be all zeros

# 4. Visualize Distributions for snp_df
print("\nVisualizing Distributions for SNP Rates:")
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1) # 1 row, 2 columns, 1st plot
sns.histplot(snp_df['Heterozygosity_Rate'], kde=True)
plt.title('Distribution of SNP Heterozygosity Rates')
plt.xlabel('Heterozygosity Rate')
plt.ylabel('Count')

plt.subplot(1, 2, 2) # 1 row, 2 columns, 2nd plot
sns.histplot(snp_df['Missingness_Rate'], kde=True, color='salmon')
plt.title('Distribution of SNP Missingness Rates')
plt.xlabel('Missingness Rate')
plt.ylabel('Count')

plt.tight_layout()
plt.show()

# 5. Sort to find SNPs with highest/lowest rates
print("\nSNPs with Highest Heterozygosity Rate:")
```

```
print("\nSNPs with Highest Heterozygosity Rate:")
display(snp_df.sort_values(by='Heterozygosity_Rate', ascending=False).head())

print("\nSNPs with Highest Missingness Rate:")
display(snp_df.sort_values(by='Missingness_Rate', ascending=False).head())
```

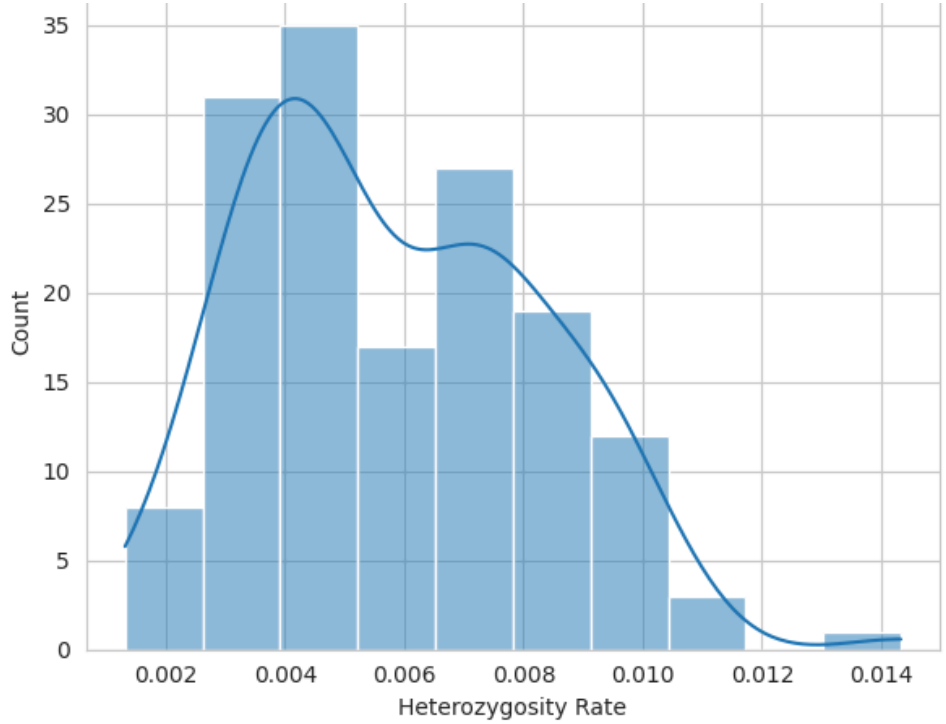
```
--- Examining sample_df ---

Sample Rates Head:
```

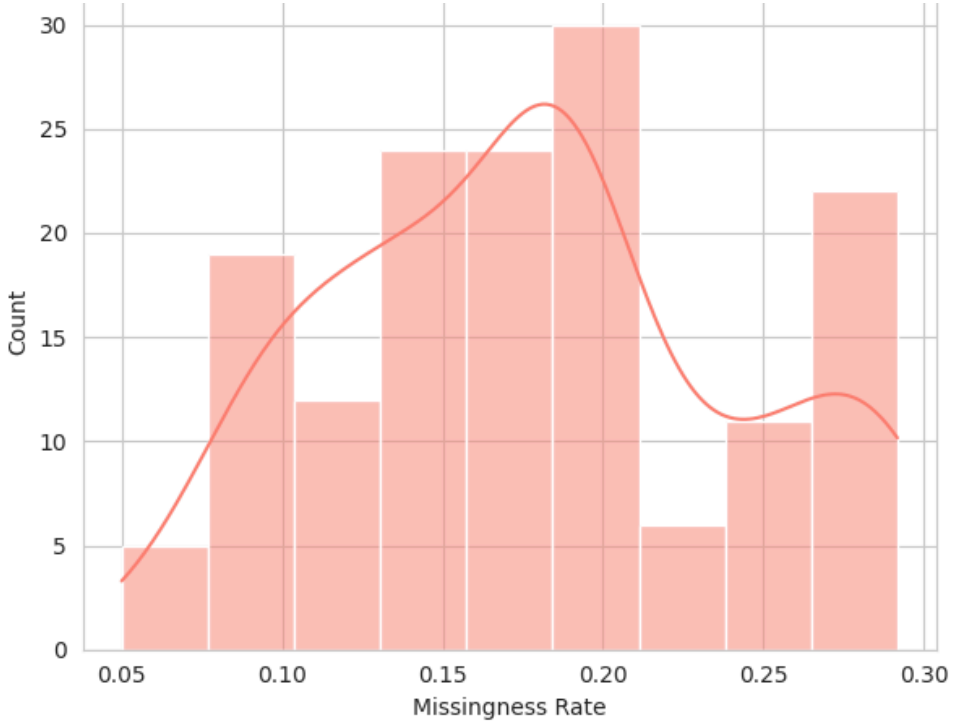
	Heterozygosity_Rate	Missingness_Rate
aculeastrum1	0.003103	0.239469
aculeatissimum1	0.002971	0.108940
aculeatissimum2	0.001915	0.097187
aethiopicum1	0.004292	0.190413
aethiopicum2	0.006140	0.194771

```
Sample Rates Summary Statistics:
```

	Heterozygosity_Rate	Missingness_Rate
count	153.000000	153.000000
mean	0.005843	0.178258
std	0.002448	0.062257
min	0.001320	0.049980
25%	0.003895	0.132378
50%	0.005414	0.180774
75%	0.007725	0.212729
max	0.014327	0.291760

```
Missing values per column in sample_df:
Heterozygosity_Rate    0
Missingness_Rate       0
dtype: int64

Visualizing Distributions for Sample Rates:
```






```
Samples with Highest Heterozygosity Rate:
```

	Heterozygosity_Rate	Missingness_Rate
macrocarpon6	0.014327	0.195563
campylacanthum4	0.010762	0.099300
dasyphyllum1	0.010762	0.284762
cerasiferum13	0.010564	0.138783
campylacanthum10	0.010366	0.095339

```
Samples with Highest Missingness Rate:
```

	Heterozygosity_Rate	Missingness_Rate
macrocarpon2	0.007329	0.291760
macrocarpon12	0.008781	0.289846

macrocarpon1	0.009045	0.289779
macrocarpon4	0.009640	0.288723
dasyphyllum5	0.007131	0.288195

```
--- Examining snp_df ---

SNP Rates Head:
```

	Heterozygosity_Rate	Missingness_Rate
0_8459	0.0	0.111111
0_8569	0.0	0.091503
0_10795	0.0	0.052288
0_10881	0.0	0.143791
0_259553	0.0	0.124183

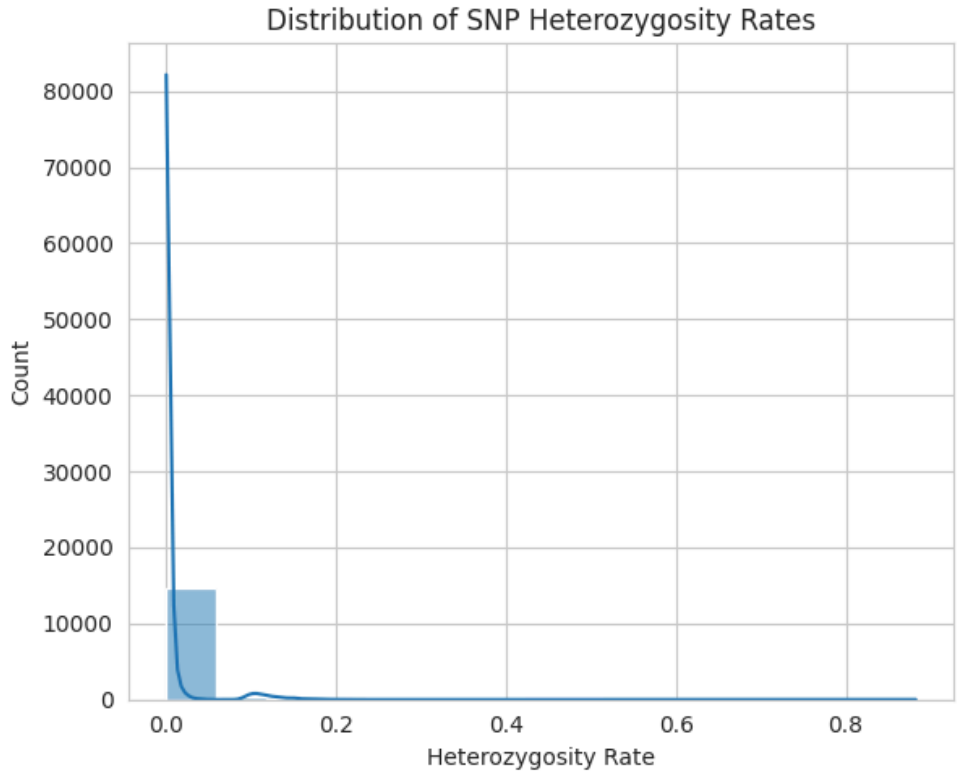
```
SNP Rates Summary Statistics:
```

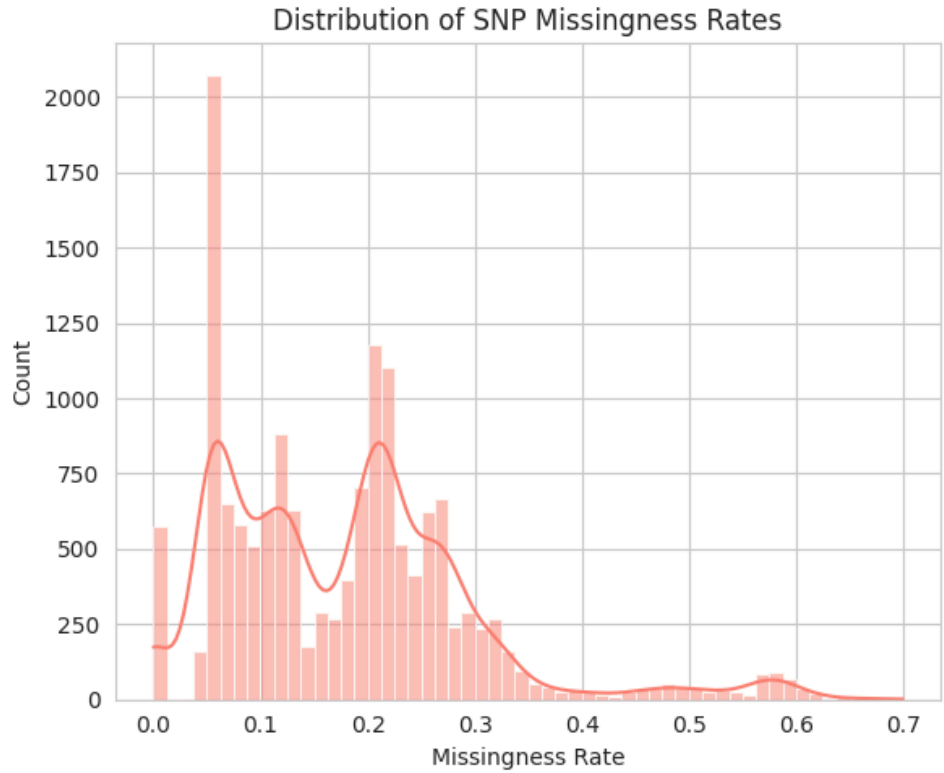
	Heterozygosity_Rate	Missingness_Rate
count	15146.000000	15146.000000
mean	0.005843	0.178258
std	0.025526	0.117589
min	0.000000	0.000000
25%	0.000000	0.084967
50%	0.000000	0.176471
75%	0.000000	0.241830
max	0.882353	0.699346

```
Missing values per column in snp_df:
Heterozygosity_Rate    0
```

```
Missingness_Rate       0
dtype: int64
```

Visualizing Distributions for SNP Rates:





SNPs with Highest Heterozygosity Rate:

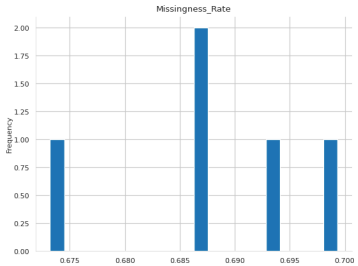
	Heterozygosity_Rate	Missingness_Rate
1_28112862	0.882353	0.0
5_60945224	0.830065	0.0
10_68840183	0.261438	0.0
8_14223303	0.228758	0.0
4_8780580	0.215686	0.0

```
SNPs with Highest Missingness Rate:
```

	Heterozygosity_Rate	Missingness_Rate
2_66148557	0.0	0.699346
4_71629926	0.0	0.692810
8_84391790	0.0	0.686275
3_86910720	0.0	0.686275
11_18301303	0.0	0.673203

```
WARNING: Runtime no longer has a reference to this dataframe, please re-run this cell and try again.
```

Distributions



Time series

Start coding or generate with AI.

```
# pca on genotype matrix plus visualisation
# pca shows us patterns or variability in a "clearer" way. It is a dimension reduction technique when you have too many observations

# Import necessary libraries
import pandas as pd
import numpy as np
from cyvcf2 import VCF
from sklearn.decomposition import PCA
import seaborn as sns
import matplotlib.pyplot as plt

# Specify the path to your VCF file
# Make sure the path is correct. If you are using the file mounted from Google Drive, use the path from the previous cells.
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
# vcf_path = '/content/Wild_African_eggplant_SNP_dataset.vcf.gz' # Use this if you downloaded it directly

# Open the VCF file
vcf = VCF(vcf_path)

# Get sample names
samples = vcf.samples
num_samples = len(samples)

# Initialize lists to store genotype data and variant positions
genotype_matrix_rows = []
variant_positions = []

# Iterate through each variant in the VCF file
for variant in vcf:
    # Create a row for the current variant's genotypes
    row = []
    # Iterate through genotypes for each sample
    # variant.genotypes returns a list of lists like [[GT_int, allele1, allele2, phasing], ...]
    for gt_info in variant.genotypes:
```

```
            # Access the genotype integers (0, 1, 2, 3 for missing)
            gt_int = gt_info[0]

            # Convert integer genotype to numerical representation
            # 0: homozygous ref (0/0), 1: heterozygous (0/1, 1/0), 2: homozygous alt (1/1), np.nan: missing
            if gt_int == 3: # Missing genotype
                row.append(np.nan)
            elif gt_int == 1: # Heterozygous
                row.append(1) # Represent heterozygous as 1
            elif gt_int == 0: # Homozygous reference
                 row.append(0) # Represent homozygous ref as 0
            elif gt_int == 2: # Homozygous alternate
                 row.append(2) # Represent homozygous alt as 2
            else:
                 # Handle unexpected genotype integers if necessary
                 row.append(np.nan)


        # Append the row of genotypes for the current variant
        genotype_matrix_rows.append(row)
        # Store the unique identifier for the variant
        variant_positions.append(f"{variant.CHROM}_{variant.POS}")

    # Close the VCF object
    vcf.close()

    # Create the genotype matrix DataFrame: Variants (rows) x Samples (columns)
    # Then transpose it to get Samples (rows) x Variants (columns) for PCA
    # PCA is typically run with observations (samples) as rows and features (variants) as columns
    G = pd.DataFrame(genotype_matrix_rows, index=variant_positions, columns=samples).T

    print("Original Genotype Matrix (Samples x Variants):")
    display(G.head())
    print(f"Shape of original matrix: {G.shape}")

    # Mean impute missing values
    # Replace NaN values with the mean of the respective variant (column)
```

```
G_imputed = G.fillna(G.mean(axis=0))

print("\nImputed Genotype Matrix Head:")
display(G_imputed.head())
print(f"Shape of imputed matrix: {G_imputed.shape}")
print(f"Number of missing values after imputation: {G_imputed.isnull().sum().sum()}") # Should be 0

# Run PCA
# Initialize PCA object, specifying the number of components (e.g., 10)
pca = PCA(n_components=10)
# Fit PCA to the imputed matrix and transform the data
pca_result = pca.fit_transform(G_imputed)

print(f"\nShape of PCA result: {pca_result.shape}")

# Create a DataFrame for the PCA results, including sample names
# We will use the first two principal components (PC1 and PC2) for visualization
pca_df = pd.DataFrame(pca_result[:, :2], columns=["PC1", "PC2"])
# Add sample names as a column to the PCA DataFrame
pca_df["sample"] = G_imputed.index # Use index from the imputed matrix (sample names)


# Map samples to species groups for coloring the PCA plot
# Define the function to extract the species group by removing trailing digits
def extract_group(sample):
    """
    Extracts a group name from a sample string by removing trailing digits.

    Args:
        sample (str): The sample name string (e.g., 'SampleA1', 'SampleB05').

    Returns:
        str: The extracted group name (e.g., 'SampleA', 'SampleB').
    """
    # Find the index of the last non-digit character from the right
    i = len(sample) - 1
    while i >= 0 and sample[i].isdigit():
```

```
            i -= 1
        # Return the substring up to (and including) the last non-digit character
        return sample[:i+1]

    # Create the species map using the extract_group function on sample names
    species_map = {s: extract_group(s) for s in samples}
    # Add the species column to the PCA DataFrame by mapping sample names
    pca_df["species"] = pca_df["sample"].map(species_map)

    print("\nPCA DataFrame Head (with species):")
    display(pca_df.head())


    # Visualize PCA with points colored by species group
    plt.figure(figsize=(10, 6))
    # Create a scatter plot using seaborn
    # x=PC1, y=PC2, color points based on 'species' column
    sns.scatterplot(data=pca_df, x="PC1", y="PC2", hue="species", palette="Set2", s=60)
    plt.title("PCA of Genotypes Colored by Species")
    plt.xlabel(f"PC1 ({pca.explained_variance_ratio_[0]:.2f}% variance explained)") # Add variance explained
    plt.ylabel(f"PC2 ({pca.explained_variance_ratio_[1]:.2f}% variance explained)") # Add variance explained
    # Place the legend outside the plot area to avoid overlapping points
    plt.legend(title="Species", bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(True, linestyle='--', alpha=0.6) # Add a subtle grid
    plt.tight_layout() # Adjust layout to prevent overlapping elements
    plt.show()

    # Optional: Print explained variance ratio for the first few components
    print("\nExplained variance ratio for the first 10 principal components:")
    print(pca.explained_variance_ratio_)
    print(f"Total variance explained by PC1 and PC2: {(pca.explained_variance_ratio_[0] + pca.explained_variance_ratio_[1]):.2f}%")
```

```
Original Genotype Matrix (Samples x Variants):
```

	0_8459	0_8569	0_10795	0_10881	0_259553	0_259625	0_1553460	0_1947362	0_1947420	0_2031644	...	12_754722
aethiopicum1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	...	1
aethiopicum2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	...	1
aethiopicum3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	...	1
aethiopicum4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	1
macrocarpon1	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	...	0

5 rows × 15146 columns

```
Shape of original matrix: (153, 15146)

Imputed Genotype Matrix Head:
```

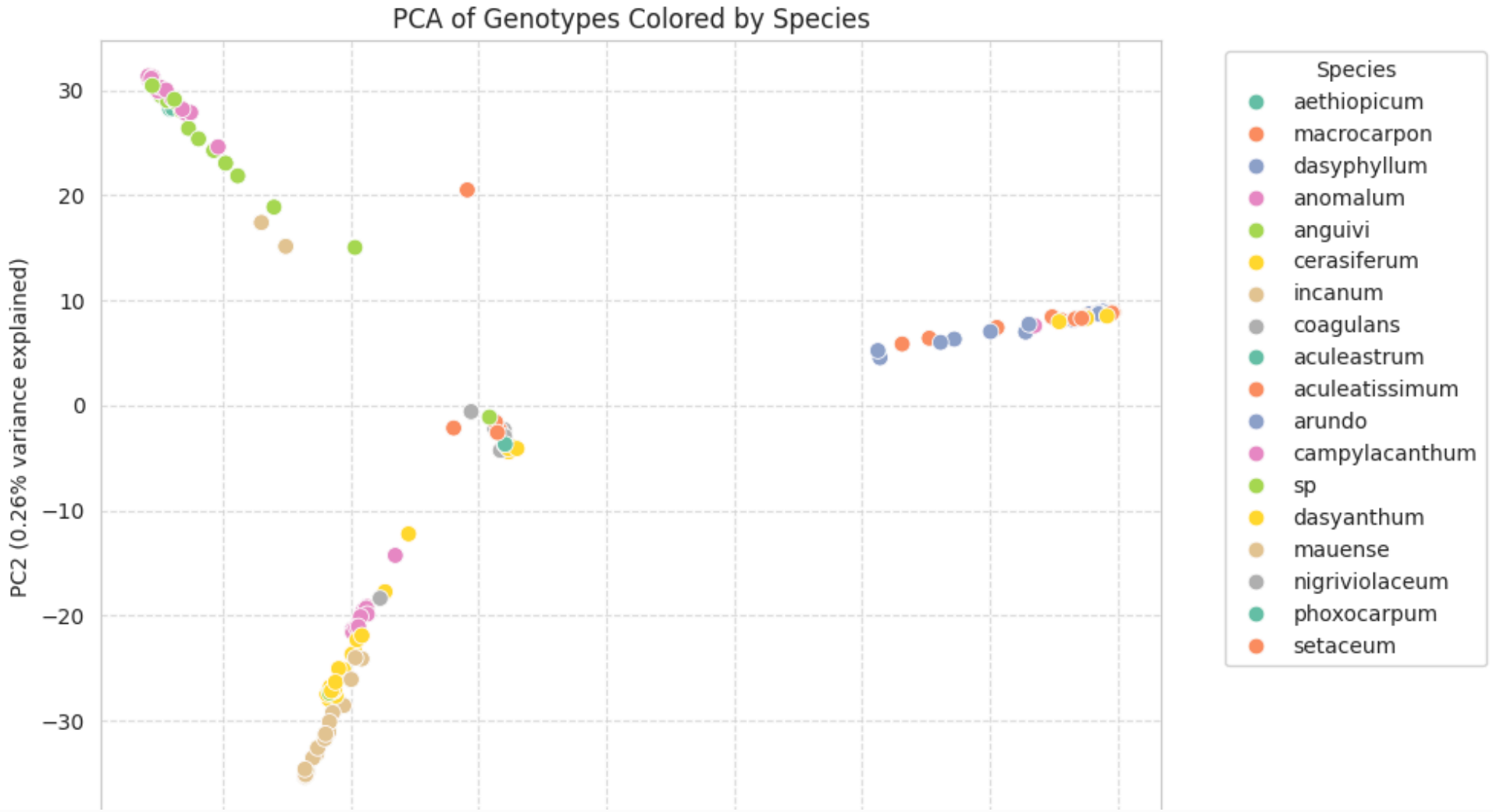
	0_8459	0_8569	0_10795	0_10881	0_259553	0_259625	0_1553460	0_1947362	0_1947420	0_2031644	...	12_754722
aethiopicum1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	...	1
aethiopicum2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	...	1
aethiopicum3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	...	1
aethiopicum4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	1
macrocarpon1	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	...	0

5 rows × 15146 columns

```
Shape of imputed matrix: (153, 15146)
Number of missing values after imputation: 0

Shape of PCA result: (153, 10)

PCA DataFrame Head (with species):
```

	PC1	PC2	sample	species
0	-24.951358	30.187278	aethiopicum1	aethiopicum
1	-25.544058	30.794733	aethiopicum2	aethiopicum

2	-24.146870	28.214889	aethiopicum3	aethiopicum
3	-23.929385	28.257496	aethiopicum4	aethiopicum
4	49.260581	9.029413	macrocarpon1	macrocarpon



```
#hypothesis testing
import numpy as np
import pandas as pd
from statsmodels.stats.multitest import multipletests
from cyvcf2 import VCF
from scipy.stats import chi2_contingency, fisher_exact
from collections import defaultdict

# Re-open the VCF file as it might have been closed in previous steps
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'
vcf = VCF(vcf_path)

samples = vcf.samples
num_samples = len(samples)

# Define your groups based on the PCA plot or known species information
# Example: Manually define two groups based on previous understanding or PCA clusters
# Replace with your actual sample names for each group
# You could potentially automate this based on clustering results from PCA
group1_name = "GroupA"
group2_name = "GroupB"

# Example mapping of samples to groups - Replace with your actual logic
# For this example, let's use the 'extract_group' function from the PCA section
# You might need to redefine it if not already in the current session
def extract_group(sample):
    """
    Extracts a group name from a sample string by removing trailing digits.
    """
    i = len(sample) - 1
    while i >= 0 and sample[i].isdigit():
        i -= 1
    return sample[:i+1]

# Create a dictionary mapping sample names to their group
# Filter samples into your chosen two groups
```

```
    sample_to_group = {s: extract_group(s) for s in samples}

    # Let's pick two specific groups identified from sample names/PCA for this example
    # You should replace 'macrocarpon' and 'aethiopicum' with the group names relevant to your analysis
    group1_samples = [s for s, group in sample_to_group.items() if group == 'macrocarpon']
    group2_samples = [s for s, group in sample_to_group.items() if group == 'aethiopicum']

    # Ensure both groups have samples
    if not group1_samples or not group2_samples:
        print("Error: One or both selected groups are empty. Please check your group definitions.")
        # Exit or handle the error appropriately
    else:
        print(f"Comparing {len(group1_samples)} samples in '{group1_name}' and {len(group2_samples)} samples in '{group2_name}'.")

        # Get the indices of samples belonging to each group
        sample_indices = {sample: i for i, sample in enumerate(samples)}
        group1_indices = [sample_indices[s] for s in group1_samples]
        group2_indices = [sample_indices[s] for s in group2_samples]

        # Lists to store results
        variant_ids = []
        p_values = []

        # Iterate through each variant (SNP) in the VCF file
        print("Performing hypothesis tests for each variant...")
        for variant in vcf:
            variant_id = f"{variant.CHROM}_{variant.POS}"
            variant_ids.append(variant_id)

            # Count alleles for each group
            # Allele counts: [ref_count, alt_count]
            group1_allele_counts = [0, 0]
            group2_allele_counts = [0, 0]

            # variant.gt_types: 0=HOM_REF, 1=HET, 2=HOM_ALT, 3=UNKNOWN
            # We consider biallelic sites (REF and one ALT).
            # For diploid organisms, a sample has two alleles.
```

```
    for idx in group1_indices:
        gt_int = variant.gt_types[idx]
        if gt_int == 0: # Homozygous reference (0/0)
            group1_allele_counts[0] += 2
        elif gt_int == 1: # Heterozygous (0/1 or 1/0)
            group1_allele_counts[0] += 1
            group1_allele_counts[1] += 1
        elif gt_int == 2: # Homozygous alternate (1/1)
            group1_allele_counts[1] += 2
        # Missing genotypes (gt_int == 3) are ignored for counting

    for idx in group2_indices:
        gt_int = variant.gt_types[idx]
        if gt_int == 0: # Homozygous reference (0/0)
            group2_allele_counts[0] += 2
        elif gt_int == 1: # Heterozygous (0/1 or 1/0)
            group2_allele_counts[0] += 1
            group2_allele_counts[1] += 1
        elif gt_int == 2: # Homozygous alternate (1/1)
            group2_allele_counts[1] += 2
        # Missing genotypes (gt_int == 3) are ignored for counting

    # Create a contingency table for the test
    # Rows: Alleles (Ref, Alt)
    # Columns: Groups (Group1, Group2)
    # table = [[Group1_ref_count, Group2_ref_count],
    #          [Group1_alt_count, Group2_alt_count]]

    contingency_table = np.array([
        [group1_allele_counts[0], group2_allele_counts[0]],
        [group1_allele_counts[1], group2_allele_counts[1]]
    ])

    # Perform the statistical test
    # Use chi2_contingency for sufficient counts, fisher_exact otherwise
    # A common heuristic is to use Fisher's exact test if any cell count is < 5
```

```
        if np.any(contingency_table < 5):
            # Fisher's exact test returns odds ratio and p-value
            # We are only interested in the p-value for significant difference
            odds_ratio, p_value = fisher_exact(contingency_table)
        else:
            # Chi-square test returns chi2 statistic, p-value, degrees of freedom, and expected frequencies
            chi2_stat, p_value, dof, expected = chi2_contingency(contingency_table)

        p_values.append(p_value)

    # Close the VCF object
    vcf.close()

    # Apply Multiple Testing Correction (Benjamini-Hochberg)
    # This is recommended to control the False Discovery Rate (FDR)
    print("\nApplying multiple testing correction...")
    reject, corrected_p_values, _, _ = multipletests(p_values, method='fdr_bh') # FDR correction

    # Create a DataFrame to store results
    test_results_df = pd.DataFrame({
        'variant_id': variant_ids,
        'p_value': p_values,
        'corrected_p_value': corrected_p_values,
        'significant_FDR_BH': reject # Boolean indicating if significant after correction
    })

    # Sort by corrected p-value to see the most significant variants first
    test_results_df = test_results_df.sort_values(by='corrected_p_value')

    # Display the first few rows of the results
    print("\nTest Results (first 10 rows, sorted by corrected p-value):")
    display(test_results_df.head(10))

    # ## Find significant SNPs and learn about the top ones

    # Define a significance threshold (e.g., alpha = 0.05)
    alpha = 0.05
```

```
# Filter for significant SNPs based on corrected p-value
significant_snps_df = test_results_df[test_results_df['corrected_p_value'] < alpha].copy()

print(f"\nNumber of significant SNPs (FDR adjusted p < {alpha}): {len(significant_snps_df)}")

if not significant_snps_df.empty:
    # Get the top 5 or 10 significant SNPs
    top_n = 10
    top_significant_snps = significant_snps_df.head(top_n)

    print(f"\nTop {top_n} significant SNPs:")
    display(top_significant_snps)

    # To find out more about these top SNPs, you would typically
    # need to re-access the VCF file or a related annotation source
    # and extract information from the INFO field for these specific variants.
    # This requires iterating through the VCF again and checking if the variant ID
    # matches one of the top significant SNPs.

    print(f"\nExtracting INFO field details for the top {top_n} significant SNPs:")
    top_snp_ids = set(top_significant_snps['variant_id'])
    vcf_info = VCF(vcf_path) # Re-open VCF to get variant info

    top_snp_details = []

    for variant in vcf_info:
        variant_id = f"{variant.CHROM}_{variant.POS}"
        if variant_id in top_snp_ids:
            # Extract desired information from the INFO field
            # The available INFO fields depend on your VCF file header
            # Common ones include AF (Allele Frequency), DP (Total Depth), NS (Number of Samples)
            # Check your VCF header (as shown in previous steps) for available INFO fields
            info_dict = dict(variant.INFO) # Convert INFO object to dictionary

            details = {
                'variant_id': variant_id,
```

```
                'CHROM': variant.CHROM,
                'POS': variant.POS,
                'REF': variant.REF,
                'ALT': variant.ALT
            }
            # Add relevant INFO fields if they exist in the VCF header
            # You need to know the INFO field IDs from your VCF header
            if 'AF' in info_dict:
                 details['INFO_AF'] = info_dict['AF']
            if 'DP' in info_dict:
                 details['INFO_DP'] = info_dict['DP']
            if 'NS' in info_dict:
                 details['INFO_NS'] = info_dict['NS']
            # Add other INFO fields you find relevant

            top_snp_details.append(details)

    vcf_info.close() # Close VCF file

    # Create a DataFrame for top SNP details
    top_snp_details_df = pd.DataFrame(top_snp_details)
    # Merge with test results to include p-values
    top_snp_details_df = pd.merge(top_snp_details_df, top_significant_snps, on='variant_id')
    top_snp_details_df = top_snp_details_df.sort_values(by='corrected_p_value') # Keep sorted by significance

    display(top_snp_details_df)

    # You can now use the `significant_snps_df` DataFrame for downstream analyses.
    # It contains the 'variant_id', 'p_value', and 'corrected_p_value' for all SNPs
    # that were found to be statistically significant.

else:
    print("No variants found to be statistically significant at the specified alpha level after correction.")
```

```
Comparing 16 samples in 'GroupA' and 4 samples in 'GroupB'.
Performing hypothesis tests for each variant...

Applying multiple testing correction...

Test Results (first 10 rows, sorted by corrected p-value):
```

	variant_id	p_value	corrected_p_value	significant_FDR_BH
10568	8_76527392	0.000023	0.011166	True
6791	5_4488301	0.000023	0.011166	True
8842	6_96394884	0.000023	0.011166	True
13274	10_86126726	0.000022	0.011166	True
9038	7_4375433	0.000014	0.011166	True
2082	1_106129098	0.000008	0.011166	True
7774	6_15616826	0.000008	0.011166	True
13157	10_83913339	0.000023	0.011166	True
8675	6_94019677	0.000023	0.011166	True
2603	2_56666321	0.000023	0.011166	True

```
Number of significant SNPs (FDR adjusted p < 0.05): 852

Top 10 significant SNPs:
```

	variant_id	p_value	corrected_p_value	significant_FDR_BH
10568	8_76527392	0.000023	0.011166	True
6791	5_4488301	0.000023	0.011166	True
8842	6_96394884	0.000023	0.011166	True
13274	10_86126726	0.000022	0.011166	True
9038	7_4375433	0.000014	0.011166	True

2082	1_106129098	0.000008	0.011166	True
7774	6_15616826	0.000008	0.011166	True
13157	10_83913339	0.000023	0.011166	True
8675	6_94019677	0.000023	0.011166	True
2603	2_56666321	0.000023	0.011166	True

```
Extracting INFO field details for the top 10 significant SNPs:
```

	variant_id	CHROM	POS	REF	ALT	INFO_AF	INFO_DP	INFO_NS	p_value	corrected_p_value	significant_FDR_BH
0	1_106129098	1	106129098	C	[T]	0.133	194	241	0.000008	0.011166	True
1	2_56666321	2	56666321	G	[C]	0.123	197	244	0.000023	0.011166	True
2	5_4488301	5	4488301	C	[G]	0.115	285	252	0.000023	0.011166	True
3	6_15616826	6	15616826	G	[C]	0.305	260	243	0.000008	0.011166	True
4	6_94019677	6	94019677	C	[T]	0.155	184	226	0.000023	0.011166	True
5	6_96394884	6	96394884	T	[C]	0.152	177	231	0.000023	0.011166	True
6	7_4375433	7	4375433	C	[T]	0.060	211	250	0.000014	0.011166	True
7	8_76527392	8	76527392	G	[A]	0.236	155	263	0.000023	0.011166	True
8	10_83913339	10	83913339	G	[A]	0.150	228	230	0.000023	0.011166	True
9	10_86126726	10	86126726	A	[G]	0.230	155	187	0.000022	0.011166	True

```
from google.colab import sheets
sheet = sheets.InteractiveSheet(df=top_significant_snps)
```

https://docs.google.com/spreadsheets/d/1u1UppQ4khpqG3RR_88nDSqIeIv2uQ02S54fnXUCA_kM/edit#gid=0

File Edit View Insert Format Data Tools Extensions Help

Menus 100% $ % .0 .00 123 Defaul... 10 B I A

A1 variant_id

	A	B	C	D	E	F	G	H	I
1	variant_id	p_value	corrected_p_val	significant_FDR_BH					
2	8_76527392	0.000022852964	0.01116551591	TRUE					
3	5_4488301	0.000022852964	0.01116551591	TRUE					
4	6_96394884	0.000022852964	0.01116551591	TRUE					
5	10_86126726	0.000021562877	0.01116551591	TRUE					
6	7_4375433	0.000014478275	0.01116551591	TRUE					
7	1_106129098	0.000007938398	0.01116551591	TRUE					
8	6_15616826	0.000007938398	0.01116551591	TRUE					
9	10_83913339	0.000022852964	0.01116551591	TRUE					
10	6_94019677	0.000022852964	0.01116551591	TRUE					
11	2_56666321	0.000022852964	0.01116551591	TRUE					
12									
13									
14									
15									
16									
17									
18									
19									
20									

Convert to table

Sheet1

Start coding or generate with AI.

```
# Step 4: Machine Learning

# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve, auc, classification_report
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
import seaborn as sns
from cyvcf2 import VCF # Needed again to get genotype data for significant SNPs
import pickle # To save and load the model and encoder (optional)

# Assuming 'significant_snps_df' from the previous step is available
# This DataFrame contains 'variant_id', 'p_value', 'corrected_p_value', 'significant_FDR_BH'
# We need to filter the *original* genotype data (or re-read from VCF)
# to create a matrix of Samples x Significant SNPs.

# Define the path to the original VCF file
vcf_path = '/content/drive/MyDrive/Colab Notebooks/Wild_African_eggplant_SNP_dataset.vcf'

# Get the list of significant SNP IDs
# Check if the DataFrame exists and is not empty
if 'significant_snps_df' in locals() and not significant_snps_df.empty:
    significant_snp_ids = set(significant_snps_df['variant_id'])
    print(f"Found {len(significant_snp_ids)} significant SNPs for machine learning.")
else:
    print("No significant SNPs found or 'significant_snps_df' is not available.")
    print("Please ensure the previous hypothesis testing step was run successfully and identified significant SNPs.")
    # If no significant SNPs, we cannot proceed with ML based on them.
    # You might choose to stop execution here or handle it differently.
    raise ValueError("Cannot proceed with ML: No significant SNPs identified.")
```

```
# --- Task: Prepare feature matrix: samples × selected SNPs; the dataframe ---

# Re-open the VCF file to extract genotypes for the significant SNPs
vcf = VCF(vcf_path)
samples = vcf.samples

# Initialize lists to store genotype data for significant SNPs
ml_genotype_matrix_rows = []
ml_variant_positions = []

# Iterate through each variant in the VCF file
print("Extracting genotypes for significant SNPs...")
for variant in vcf:
    variant_id = f"{variant.CHROM}_{variant.POS}"

    # Check if the current variant is in our list of significant SNPs
    if variant_id in significant_snp_ids:
        ml_variant_positions.append(variant_id)

        # Create a row for the current variant's genotypes, converted to numerical
        row = []
        # variant.genotypes is a list of lists like [[GT_int, allele1, allele2, phasing], ...]
        for gt_info in variant.genotypes:
             gt_int = gt_info[0]
             # Convert integer genotype to numerical representation (0, 1, 2, NaN)
             if gt_int == 3: # Missing genotype
                 row.append(np.nan)
             elif gt_int == 1: # Heterozygous
                 row.append(1)
             elif gt_int == 0: # Homozygous reference
                  row.append(0)
             elif gt_int == 2: # Homozygous alternate
                  row.append(2)
             else:
                  row.append(np.nan) # Should not happen with 0,1,2,3
```

```
            ml_genotype_matrix_rows.append(row)

    # Close the VCF object
    vcf.close()

    # Create the feature matrix DataFrame: Significant Variants (rows) x Samples (columns)
    # Then transpose it to get Samples (rows) x Significant Variants (columns)
    if not ml_variant_positions or not ml_genotype_matrix_rows:
         raise ValueError("No genotype data extracted for significant SNPs.")

    feature_matrix = pd.DataFrame(ml_genotype_matrix_rows, index=ml_variant_positions, columns=samples).T

    print("\nFeature Matrix (Samples x Significant SNPs) Head:")
    display(feature_matrix.head())
    print(f"Shape of feature matrix: {feature_matrix.shape}")

    # Handle missing values in the feature matrix
    # Imputation is necessary for most ML models
    # Using mean imputation as a simple approach. Store the means for later use on unseen data.
    feature_means = feature_matrix.mean(axis=0)
    feature_matrix_imputed = feature_matrix.fillna(feature_means)
    print(f"\nNumber of missing values after imputation: {feature_matrix_imputed.isnull().sum().sum()}") # Should be 0

    # --- Task: Encode group labels (e.g., 0 = macrocarpon, 1 = dasyphyllum, etc.) ---

    # Use the sample_to_group mapping created in the hypothesis testing step
    if 'sample_to_group' in locals():
        # Get the species labels for the samples in the feature matrix
        # Ensure the samples in the feature matrix are mapped correctly
        sample_names_in_matrix = feature_matrix_imputed.index
        sample_labels = pd.Series([sample_to_group.get(sample, np.nan) for sample in sample_names_in_matrix], index=sample_names_in_matr


        # Handle samples with missing or unknown group labels
        if sample_labels.isnull().any():
            print("Warning: Some samples in the feature matrix do not have a group label or their group was not in the original sample_t
            print("Samples with missing labels will be excluded.")
```

```
    # Identify samples with missing labels
    samples_to_exclude = sample_labels[sample_labels.isnull()].index
    feature_matrix_imputed = feature_matrix_imputed.drop(samples_to_exclude)
    sample_labels = sample_labels.drop(samples_to_exclude)
    print(f"Excluded {len(samples_to_exclude)} samples with missing group labels.")


# Check group sizes before encoding and splitting
group_counts = sample_labels.value_counts()
print("\nCounts per original group:")
print(group_counts)

# Identify groups with only one member
groups_to_remove = group_counts[group_counts < 2].index
if not groups_to_remove.empty:
    print(f"\nRemoving samples from groups with fewer than 2 members for stratification: {list(groups_to_remove)}")
    samples_to_remove = sample_labels[sample_labels.isin(groups_to_remove)].index
    feature_matrix_imputed = feature_matrix_imputed.drop(samples_to_remove)
    sample_labels = sample_labels.drop(samples_to_remove)
    print(f"Excluded {len(samples_to_remove)} samples from small groups.")
    # Re-check group counts
    print("\nCounts per group after removal:")
    print(sample_labels.value_counts())


if feature_matrix_imputed.empty or sample_labels.empty:
     raise ValueError("Feature matrix or sample labels are empty after handling small groups/missing labels.")

# Encode the categorical group labels into numerical labels
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(sample_labels)

print("\nEncoded Group Labels (first 10):")
print(encoded_labels[:10])
print("\nOriginal Group Names and their Encoded Values:")
# Create a mapping back from encoded label to original group name
unique_encoded_labels = np.unique(encoded_labels)
```

```
unique_group_names = label_encoder.inverse_transform(unique_encoded_labels)
print(dict(zip(unique_encoded_labels, unique_group_names)))


# --- Task: Train a simple classifier ---

# Define features (X) and target (y) using the filtered data
X = feature_matrix_imputed
y = encoded_labels

# Check if we still have multiple classes and enough samples for splitting
if len(np.unique(y)) < 2:
     raise ValueError("Only one class remaining after filtering. Cannot perform classification.")
if len(y) < 2: # Need at least 2 samples for splitting
     raise ValueError("Fewer than 2 samples remaining after filtering. Cannot perform classification.")

# Split data into training and testing sets
# Using stratify=y ensures that the proportion of classes in the training
# and testing sets is the same as in the original dataset (now that small classes are handled).
# Ensure test_size is appropriate given the number of samples in the smallest remaining class.
# A test_size of 0.25 (25%) is generally fine, but if you have very few samples overall,
# you might need to adjust it or use cross-validation.
try:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42, stratify=y)
    print(f"\nTraining data shape: {X_train.shape}, Testing data shape: {X_test.shape}")
except ValueError as e:
    print(f"Error during train_test_split: {e}")
    print("This might still be due to a small class size or the chosen test_size.")
    print("Consider adjusting test_size or examining the class counts again.")
    raise # Re-raise the error


# Initialize and train a Random Forest Classifier
# RandomForestClassifier is a good choice for this type of data
model = RandomForestClassifier(n_estimators=100, random_state=42) # n_estimators is number of trees
model.fit(X_train, y_train)
```

```
print("\nRandom Forest Classifier trained successfully.")


# --- Task: Evaluate with accuracy, confusion matrix, and maybe ROC curve ---

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

# Get the class labels present in y_test
test_class_labels = np.unique(y_test)
test_target_names = label_encoder.inverse_transform(test_class_labels)

class_report = classification_report(y_test, y_pred, target_names=test_target_names)

print(f"\nModel Accuracy on Test Set: {accuracy:.4f}")
print("\nConfusion Matrix:")
# Display confusion matrix with labels
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
        xticklabels=test_target_names, yticklabels=test_target_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

print("\nClassification Report:")
print(class_report)


# ROC Curve (only applicable for binary classification)
# Check if it's binary classification based on the unique labels *after* filtering
if len(np.unique(y)) == 2:
     print("\nROC Curve (Binary Classification):")
```

```
        # Get predicted probabilities for the positive class (class 1, based on sorted unique labels)
        # Ensure the class with encoded label 1 is used
        y_prob = model.predict_proba(X_test)[:, 1]
        fpr, tpr, thresholds = roc_curve(y_test, y_prob)
        roc_auc = auc(fpr, tpr)

        plt.figure(figsize=(8, 6))
        plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
        plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('Receiver Operating Characteristic (ROC) Curve')
        plt.legend(loc="lower right")
        plt.show()
    elif len(np.unique(y)) > 2:
        print("\nROC Curve is typically visualized for binary classification.")
        print("For multi-class, consider plotting ROC curves for each class (OvR) or using other metrics.")
    else:
        print("\nOnly one class found after filtering, cannot plot ROC curve.")


    # --- Task: Check feature importance ---
    # For tree-based models like Random Forest, feature importance can be assessed

    print("\nFeature Importances (Top 10 Significant SNPs):")
    # Get feature importances from the trained model
    importances = model.feature_importances_

    # Create a Series for importances with variant IDs as index
    feature_importances = pd.Series(importances, index=X_train.columns)

    # Sort importances and get the top N
    top_features = feature_importances.sort_values(ascending=False).head(10)

    display(top_features)

    # Optional: Visualize feature importances
```

```
if not top_features.empty:
    plt.figure(figsize=(10, 6))
    sns.barplot(x=top_features.values, y=top_features.index, palette='viridis')
    plt.title('Top 10 Most Important Significant SNPs')
    plt.xlabel('Importance')
    plt.ylabel('Significant SNP (Variant ID)')
    plt.tight_layout()
    plt.show()


# --- Outcome: You build a model that predicts species group based on genotype. ---
# --- Save and test your model on unseen data. ---
# Saving the model and encoder are good practice

# Example of how to save the model using pickle
model_filename = 'random_forest_species_classifier.pkl'
with open(model_filename, 'wb') as file:
    pickle.dump(model, file)
print(f"\nModel saved to {model_filename}")

# Example of how to save the LabelEncoder
encoder_filename = 'label_encoder.pkl'
with open(encoder_filename, 'wb') as file:
     pickle.dump(label_encoder, file)
print(f"LabelEncoder saved to {encoder_filename}")

# Example of how to save the feature means (for imputing unseen data)
feature_means_filename = 'feature_means.pkl'
with open(feature_means_filename, 'wb') as file:
    pickle.dump(feature_means, file)
print(f"Feature means saved to {feature_means_filename}")

# To test on unseen data, you would:
# 1. Load the saved model, encoder, and feature means.
# 2. Prepare the unseen data: load its VCF, extract genotypes for the *same* significant SNPs used for training.
#    Ensure the unseen data matrix has the same columns (SNPs) in the same order as the training data (X_train.columns).
# 3. Impute missing values in the unseen data using the *loaded feature means* (not the mean of the unseen data).
```

```
        # 4. Use model.predict() on the prepared unseen data matrix.
        # 5. Use label_encoder.inverse_transform() to convert predicted numerical labels back to original group names.


    else: # This else corresponds to the initial check for significant_snp_ids
        print("\nMachine learning step skipped because no significant SNPs were identified.")
```

```
Found 852 significant SNPs for machine learning.
Extracting genotypes for significant SNPs...

Feature Matrix (Samples x Significant SNPs) Head:
```

	0_4689655	0_5252678	0_11744338	0_17209575	0_17467663	0_41386401	1_219017	1_219087	1_311772	1_915464	..
aethiopicum1	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
aethiopicum2	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
aethiopicum3	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
aethiopicum4	0.0	0.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	0.0	
macrocarpon1	1.0	0.0	1.0	0.0	1.0	1.0	NaN	NaN	1.0	1.0	

5 rows × 852 columns

```
Shape of feature matrix: (153, 852)

Number of missing values after imputation: 0

Counts per original group:
cerasiferum       26
anomalum          22
incanum           19
macrocarpon       16
anguivi           16
dasyphyllum       15
campylacanthum    13
coagulans          6
aethiopicum        4
sp                 4
setaceum           2
aculeatissimum     2
dasyanthum         2
mauense            2
aculeastrum        1
arundo             1
nigriviolaceum     1
phoxocarpum        1
Name: count, dtype: int64
```

```
Removing samples from groups with fewer than 2 members for stratification: ['aculeastrum', 'arundo', 'nigriviolaceum', 'phoxo
Excluded 4 samples from small groups.

Counts per group after removal:
cerasiferum       26
anomalum          22
incanum           19
anguivi           16
macrocarpon       16
dasyphyllum       15
campylacanthum    13
coagulans          6
aethiopicum        4
sp                 4
aculeatissimum     2
dasyanthum         2
mauense            2
setaceum           2
Name: count, dtype: int64

Encoded Group Labels (first 10):
[ 1  1  1  1 10 10 10  8  8 10]

Original Group Names and their Encoded Values:
{np.int64(0): 'aculeatissimum', np.int64(1): 'aethiopicum', np.int64(2): 'anguivi', np.int64(3): 'anomalum', np.int64(4): 'ca

Training data shape: (111, 852), Testing data shape: (38, 852)

Random Forest Classifier trained successfully.
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))

Model Accuracy on Test Set: 0.7368

Confusion Matrix:
```

Confusion Matrix

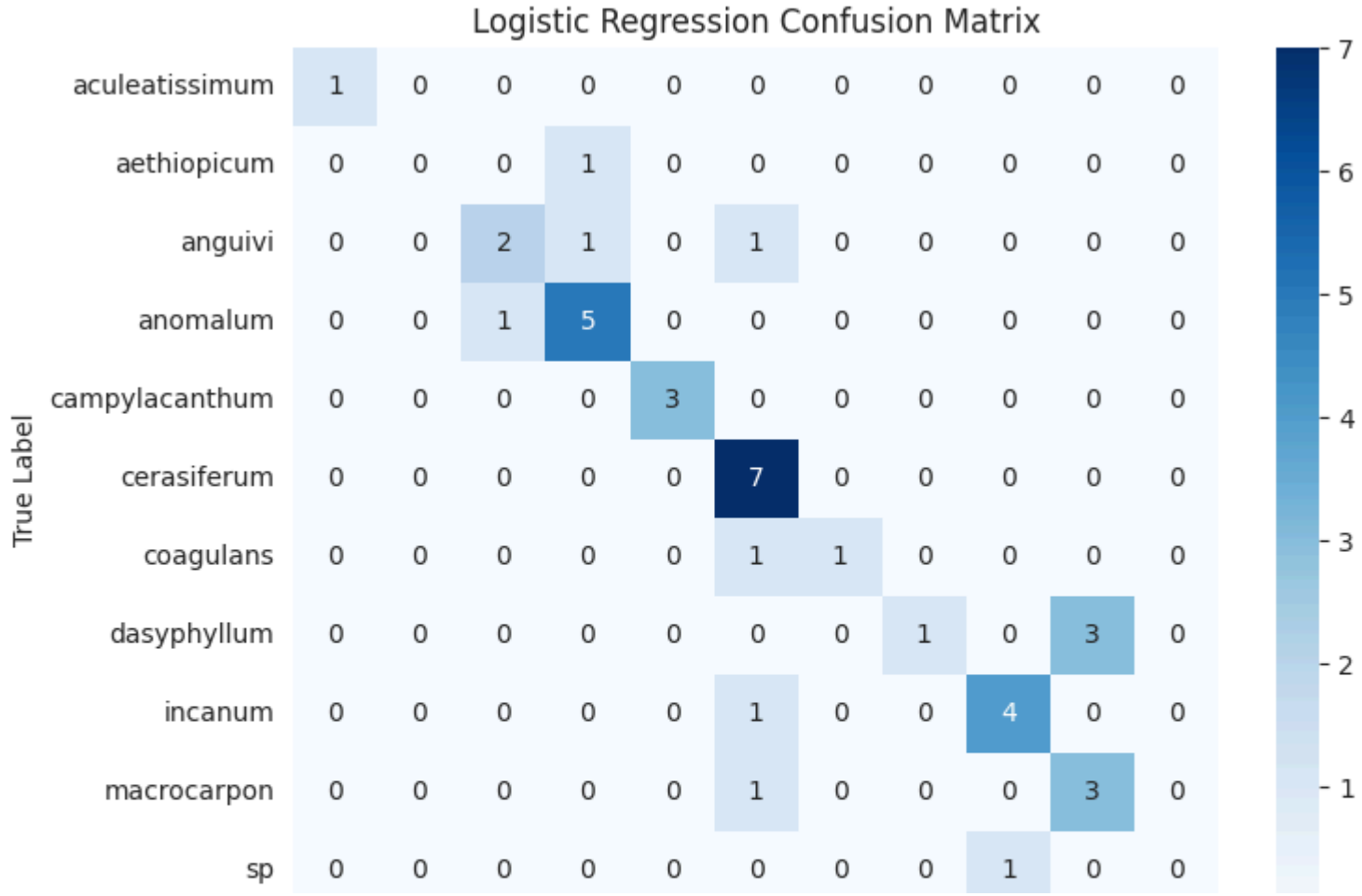
Start coding or generate with AI.

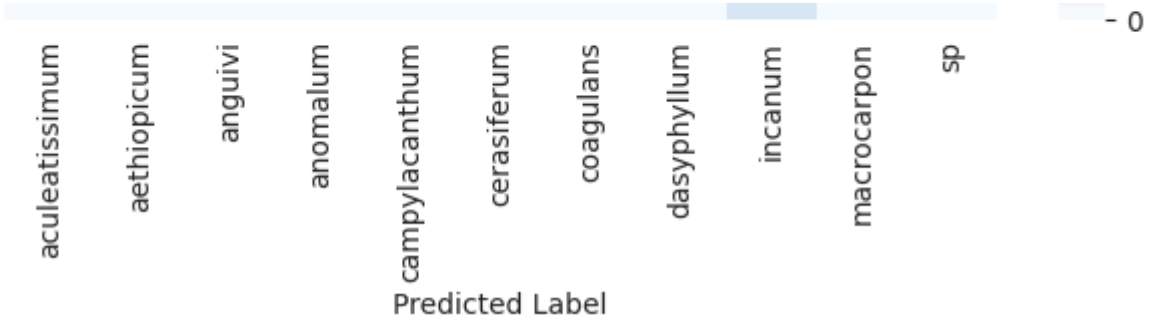
```
## Example 1: Logistic Regression Classifier

print("\n--- Training and Evaluating Logistic Regression ---")

# Import Logistic Regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Initialize the Logistic Regression model
# Increase max_iter if convergence warnings appear
# Add solver='liblinear' or other solvers if needed based on data size/type
log_reg_model = LogisticRegression(random_state=42, max_iter=1000, solver='liblinear')

# Train the model
log_reg_model.fit(X_train, y_train)

print("Logistic Regression model trained successfully.")

# Make predictions on the test set
y_pred_lr = log_reg_model.predict(X_test)

# Evaluate the model
accuracy_lr = accuracy_score(y_test, y_pred_lr)
conf_matrix_lr = confusion_matrix(y_test, y_pred_lr)
class_report_lr = classification_report(y_test, y_pred_lr, target_names=test_target_names)

print(f"\nLogistic Regression Accuracy on Test Set: {accuracy_lr:.4f}")
print("\nLogistic Regression Confusion Matrix:")
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_lr, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_target_names, yticklabels=test_target_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Logistic Regression Confusion Matrix')
```

```
plt.show()

print("\nLogistic Regression Classification Report:")
print(class_report_lr)

# ROC Curve (Binary Classification Only)
if len(np.unique(y_test)) == 2:
    print("\nLogistic Regression ROC Curve (Binary Classification):")
    y_prob_lr = log_reg_model.predict_proba(X_test)[:, 1]
    fpr_lr, tpr_lr, thresholds_lr = roc_curve(y_test, y_prob_lr)
    roc_auc_lr = auc(fpr_lr, tpr_lr)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr_lr, tpr_lr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc_lr:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Logistic Regression ROC Curve')
    plt.legend(loc="lower right")
    plt.show()

# Feature Importance (Coefficients for Logistic Regression)
# Note: Feature importance in linear models is based on coefficients, not feature_importances_
# Coefficients represent the change in the log-odds of the target variable for a one-unit change in the feature.
# Magnitude indicates importance, sign indicates direction. Need to consider potential scaling.
print("\nLogistic Regression Feature Coefficients (Top/Bottom 10):")
coefficients = log_reg_model.coef_ # This will be shape (n_classes - 1, n_features) for multi-class
# For simplicity, we can look at the magnitude of coefficients, or if binary, the single row.

if len(np.unique(y_test)) == 2: # Binary classification
    coef_series = pd.Series(coefficients[0], index=X_train.columns)
    sorted_coef = coef_series.abs().sort_values(ascending=False)
    top_coef_indices = sorted_coef.head(10).index
    top_coef_values = coef_series[top_coef_indices] # Get the actual coefficients (with signs)

    print("Top 10 features by absolute coefficient magnitude:")
    display(top_coef_values)
```

```
    plt.figure(figsize=(10, 6))
    sns.barplot(x=top_coef_values.values, y=top_coef_values.index, palette='coolwarm')
    plt.title('Top 10 Logistic Regression Feature Coefficients')
    plt.xlabel('Coefficient Value')
    plt.ylabel('Significant SNP (Variant ID)')
    plt.tight_layout()
    plt.show()

else: # Multi-class classification
    # For multi-class (OvR or Multinomial), interpreting coefficients is more complex.
    # One common approach is to look at the average absolute coefficient magnitude across classes.
    avg_abs_coef = np.mean(np.abs(coefficients), axis=0)
    coef_series = pd.Series(avg_abs_coef, index=X_train.columns)
    sorted_coef = coef_series.sort_values(ascending=False)
    top_coef_indices = sorted_coef.head(10).index
    top_coef_values = coef_series[top_coef_indices] # Get the average absolute coefficients

    print("Top 10 features by average absolute coefficient magnitude (Multi-class):")
    display(top_coef_values)

    plt.figure(figsize=(10, 6))
    sns.barplot(x=top_coef_values.values, y=top_coef_values.index, palette='viridis')
    plt.title('Top 10 Logistic Regression Feature Importance (Avg Abs Coefficient)')
    plt.xlabel('Average Absolute Coefficient Magnitude')
    plt.ylabel('Significant SNP (Variant ID)')
    plt.tight_layout()
    plt.show()
```

```
--- Training and Evaluating Logistic Regression ---
Logistic Regression model trained successfully.

Logistic Regression Accuracy on Test Set: 0.7105

Logistic Regression Confusion Matrix:
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Precision is ill-def
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```






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

aculeatissimum       1.00      1.00      1.00         1
   aethiopicum       0.00      0.00      0.00         1
       anguivi       0.67      0.50      0.57         4
      anomalum       0.71      0.83      0.77         6
campylacanthum       1.00      1.00      1.00         3
   cerasiferum       0.64      1.00      0.78         7
     coagulans       1.00      0.50      0.67         2
   dasyphyllum       1.00      0.25      0.40         4
       incanum       0.80      0.80      0.80         5
   macrocarpon       0.50      0.75      0.60         4
            sp       0.00      0.00      0.00         1

      accuracy                           0.71        38
     macro avg       0.67      0.60      0.60        38
  weighted avg       0.72      0.71      0.68        38


Logistic Regression Feature Coefficients (Top/Bottom 10):
Top 10 features by average absolute coefficient magnitude (Multi-class):
```

	0
6_91900254	0.455496
5_78458797	0.379226
5_70821000	0.302473
12_71489388	0.298195

Start coding or generate with AI.

```
## Example 2: Support Vector Machine (SVM) Classifier

print("\n--- Training and Evaluating Support Vector Machine ---")

# Import SVM
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Initialize the SVM model
# Using a linear kernel for simplicity and interpretability with genotype data.
# For non-linear relationships, try kernel='rbf' (Radial Basis Function) - but might require scaling.
svm_model = SVC(kernel='linear', probability=True, random_state=42) # probability=True is needed for ROC curve

# Train the model
svm_model.fit(X_train, y_train)

print("SVM model trained successfully.")

# Make predictions on the test set
y_pred_svm = svm_model.predict(X_test)

# Evaluate the model
accuracy_svm = accuracy_score(y_test, y_pred_svm)
conf_matrix_svm = confusion_matrix(y_test, y_pred_svm)
class_report_svm = classification_report(y_test, y_pred_svm, target_names=test_target_names)

print(f"\nSVM Accuracy on Test Set: {accuracy_svm:.4f}")
print("\nSVM Confusion Matrix:")
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_svm, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_target_names, yticklabels=test_target_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('SVM Confusion Matrix')
```

```
plt.show()

print("\nSVM Classification Report:")
print(class_report_svm)

# ROC Curve (Binary Classification Only)
if len(np.unique(y_test)) == 2:
    print("\nSVM ROC Curve (Binary Classification):")
    # Need predict_proba for ROC curve
    y_prob_svm = svm_model.predict_proba(X_test)[:, 1]
    fpr_svm, tpr_svm, thresholds_svm = roc_curve(y_test, y_prob_svm)
    roc_auc_svm = auc(fpr_svm, tpr_svm)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr_svm, tpr_svm, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc_svm:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('SVM ROC Curve')
    plt.legend(loc="lower right")
    plt.show()

# Feature Importance for SVM (based on coefficients for linear kernel)
# Similar to Logistic Regression, for linear SVM, coefficients indicate importance.
if svm_model.kernel == 'linear':
     print("\nSVM Feature Coefficients (linear kernel, Top/Bottom 10):")
     coefficients_svm = svm_model.coef_ # Shape will be (n_classes * (n_classes - 1) / 2) for 'ovo' multi-class or (n_classes, n_fea
     # SVC defaults to 'ovr' (one-vs-rest) strategy for multi-class, so coef_ shape is (n_classes, n_features)

     if len(np.unique(y_test)) == 2: # Binary classification
         coef_series_svm = pd.Series(coefficients_svm[0], index=X_train.columns)
         sorted_coef_svm = coef_series_svm.abs().sort_values(ascending=False)
         top_coef_indices_svm = sorted_coef_svm.head(10).index
         top_coef_values_svm = coef_series_svm[top_coef_indices_svm] # Get the actual coefficients (with signs)

         print("Top 10 features by absolute coefficient magnitude:")
         display(top_coef_values_svm)
```

```
        plt.figure(figsize=(10, 6))
        sns.barplot(x=top_coef_values_svm.values, y=top_coef_values_svm.index, palette='coolwarm')
        plt.title('Top 10 SVM Feature Coefficients (Linear Kernel)')
        plt.xlabel('Coefficient Value')
        plt.ylabel('Significant SNP (Variant ID)')
        plt.tight_layout()
        plt.show()

    else: # Multi-class classification ('ovr' strategy)
        # Average absolute coefficient magnitude across the one-vs-rest classifiers
        avg_abs_coef_svm = np.mean(np.abs(coefficients_svm), axis=0)
        coef_series_svm = pd.Series(avg_abs_coef_svm, index=X_train.columns)
        sorted_coef_svm = coef_series_svm.sort_values(ascending=False)
        top_coef_indices_svm = sorted_coef_svm.head(10).index
        top_coef_values_svm = coef_series_svm[top_coef_indices_svm] # Get the average absolute coefficients

        print("Top 10 features by average absolute coefficient magnitude (Linear SVM, Multi-class OvR):")
        display(top_coef_values_svm)

        plt.figure(figsize=(10, 6))
        sns.barplot(x=top_coef_values_svm.values, y=top_coef_values_svm.index, palette='viridis')
        plt.title('Top 10 SVM Feature Importance (Avg Abs Coefficient)')
        plt.xlabel('Average Absolute Coefficient Magnitude')
        plt.ylabel('Significant SNP (Variant ID)')
        plt.tight_layout()
        plt.show()
else:
    print("\nFeature importance (coefficients) is only directly interpretable for SVM with a linear kernel.")
```