lab 10 - Classification

Name: Robb Alexander and Ryan Bailis
Class: CSCI349
Semester: 2021SP
Instructor: Brian King

In [1]:
```
# Setting things up
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly
import plotly.graph_objects as go
import plotly.express as px
```

1) Read about the famous Fisher's Iris dataset. This is perhaps the most commonly used dataset to teach students how to build classification models: https://en.wikipedia.org/wiki/Iris_flower_data_set (https://en.wikipedia.org/wiki/Iris_flower_data_set) : Then, include the following code to import a copy of the Iris data from Seaborn's library of datasets: df_iris = sns.load_dataset('iris') Print out the results of info() to understand the types of each variable as read in from the csv file.

In [2]:
```
df_iris = sns.load_dataset('iris')
df_iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

2) Be thankful for a moment, because the data are clean. However, the species variable needs work. Convert the variable to a pandas Categorical variable. Then show the distribution of your variable (how many of each species?). Repeat the info() output to show that your variable is now categorical, and not merely an object.

In [3]:
```
df_iris.species = pd.Categorical(df_iris.species)
df_iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype   
---  ------        --------------  -----   
 0   sepal_length  150 non-null    float64 
 1   sepal_width   150 non-null    float64 
 2   petal_length  150 non-null    float64 
 3   petal_width   150 non-null    float64 
 4   species       150 non-null    category
dtypes: category(1), float64(4)
memory usage: 5.1 KB
```

In [4]:
```
df_iris.species.value_counts()
```

Out[4]:
```
setosa        50
versicolor    50
virginica     50
Name: species, dtype: int64
```

3) Now, perform essential summarizing tasks on your data. Show the output of describe() and show the first 10 observations.

In [5]:
```
df_iris.describe()
```

Out[5]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

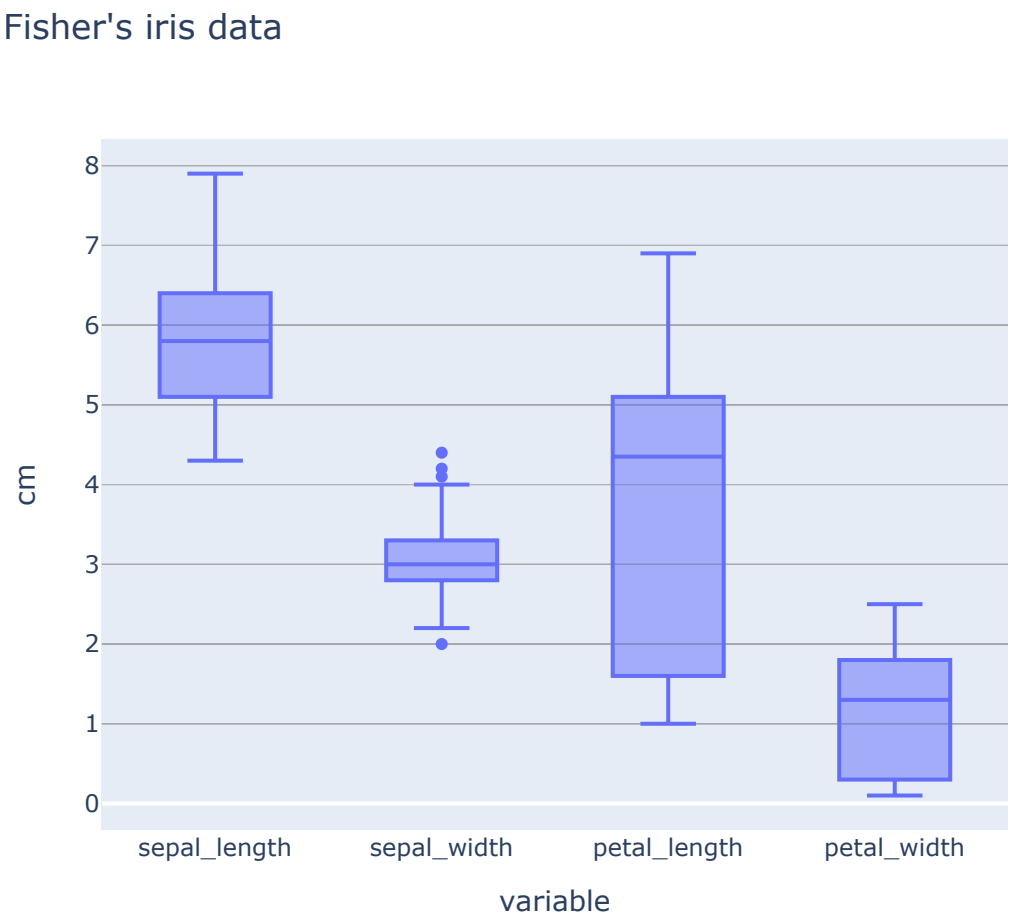
In [6]:
```
df_iris.head(10)
```

Out[6]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa

4) Always start with basic univariate plots. Create a single boxplot showing the distribution of each of the four independent variables on one plot, using a boxplot. Ignore the target variable species for the time.

In [7]:
```
px.box(df_iris.drop("species", axis=1), title="Fisher's iris data", labe
ls={"value":"cm"})
```



In [8]:
```
df_iris_melt = pd.melt(df_iris, id_vars="species")
fig = px.box(df_iris_melt, x='variable', y='value', title="Fisher's Iri
s")
```

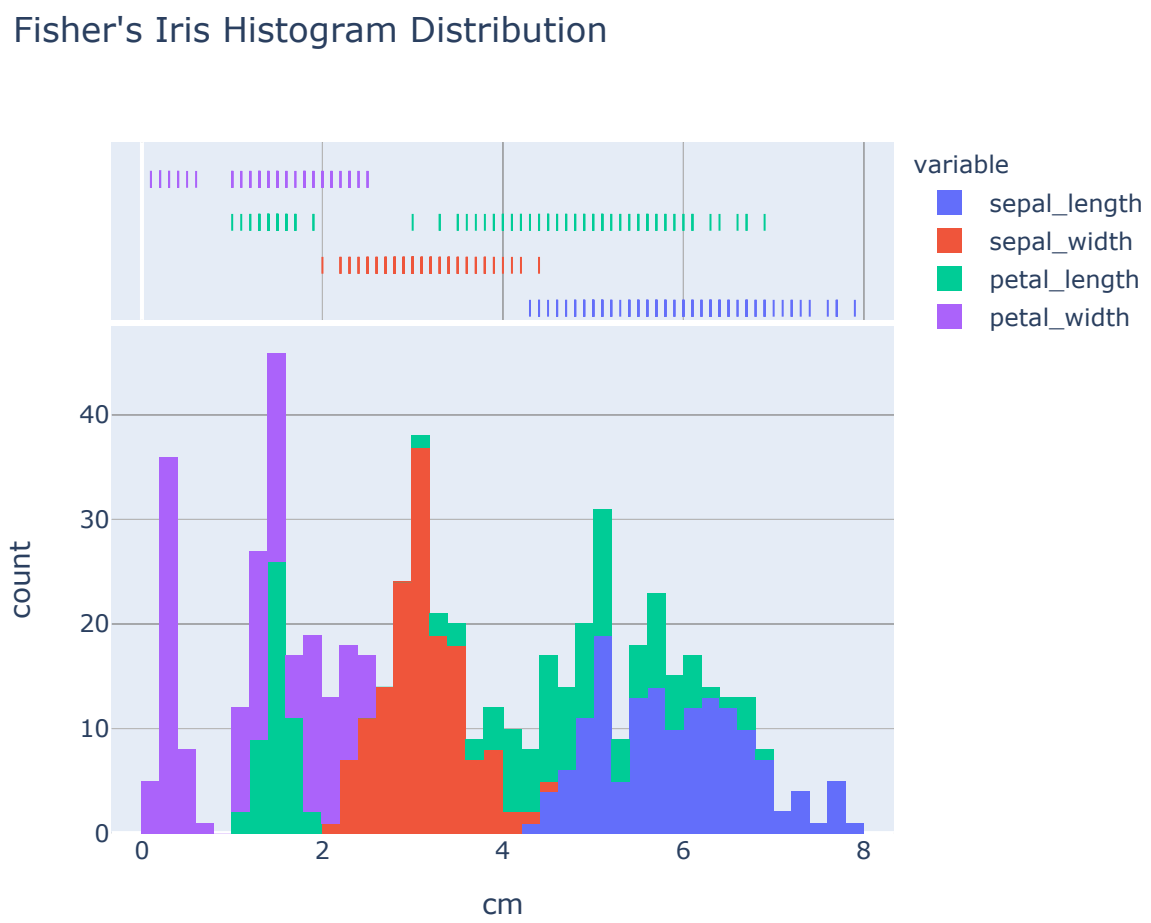
5) Violin plots are becoming increasingly common in data science. First, briefly explain what a violin plot is. Then, figure out how to generate a univariate violin plot of each independent variable. Compare and contrast your violin plot against the boxplot. (NOTE: Seaborn makes this very easy!)

Violin plots are the plots that show the distribution of the data with respect to a type of category. It is similar to the boxplot but with all the distribution rather than just the average, iqrs, min/max, outliers. It can look like a violin from the density distribution on both sides.

In [9]:
```
px.violin(df_iris_melt, x='variable', y='value', title="Fisher's Iris Violin Distribution", labels={"value": "cm"})
```

6) Show a histogram and/or a density plot of each variable on a single plot. And be sure to provide some way to see the distribution of all four variables separately. You could use alpha blending on the histogram, or perhaps consider a "rugplot" overlaid on top.

In [10]:
```
px.histogram(df_iris_melt, x="value",  color="variable", marginal="rug",
nbins=40, title="Fisher's Iris Histogram Distribution", labels={"value":
"cm"})
```

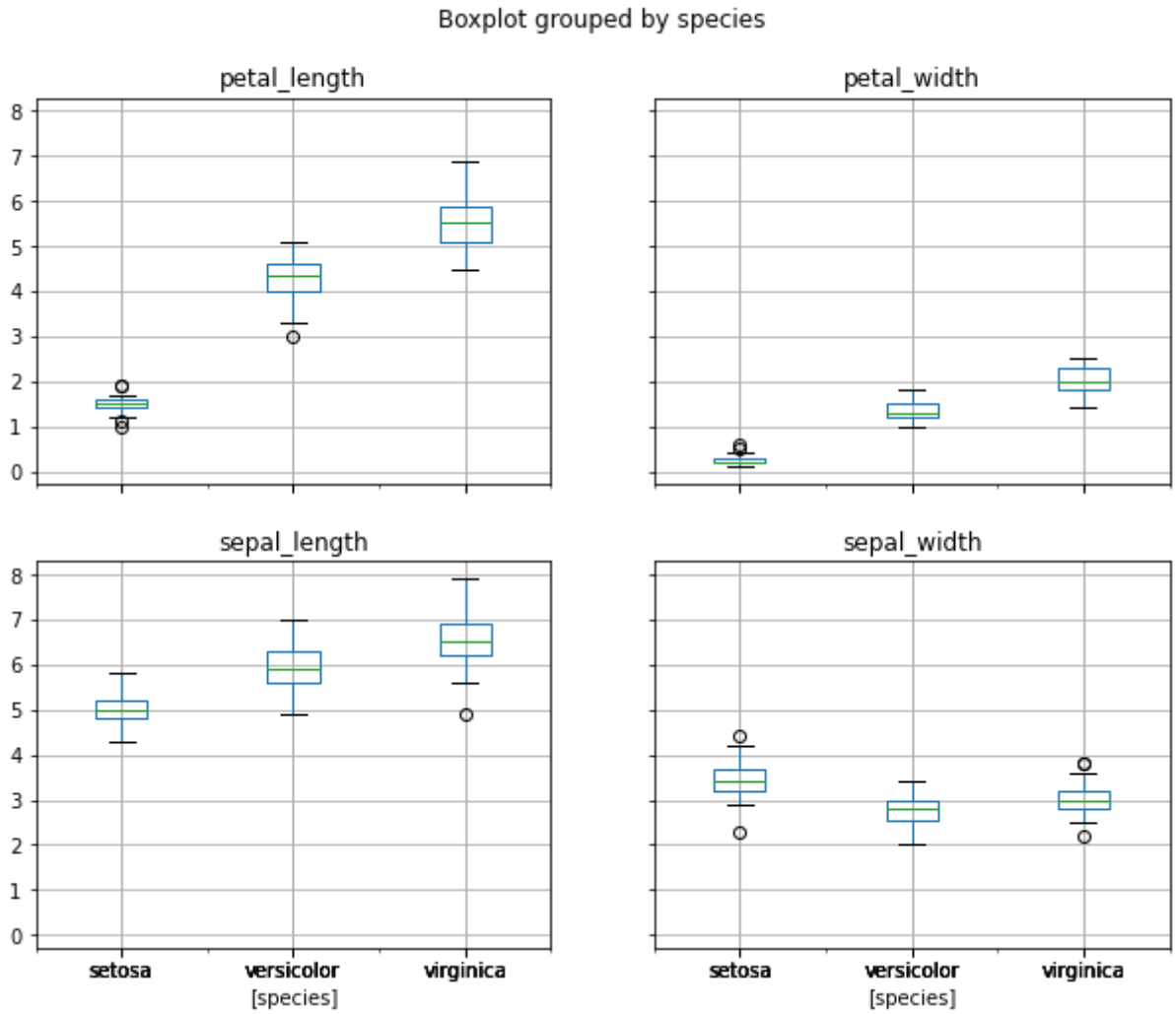


7) Summarize your findings from these plots. Is one most ideal for understanding your distribution? Characterize the distribution of your four variables. Remember, this is a univariate exploration, so you don't care about the class variable yet.

Sepal width and Sepal Length look like bad ways to classify the Irises, this is because there is no clear split in the distribution of data. From the looks of it, the Pedal Width and Length both have a clear and clean cut to allow for good classification.

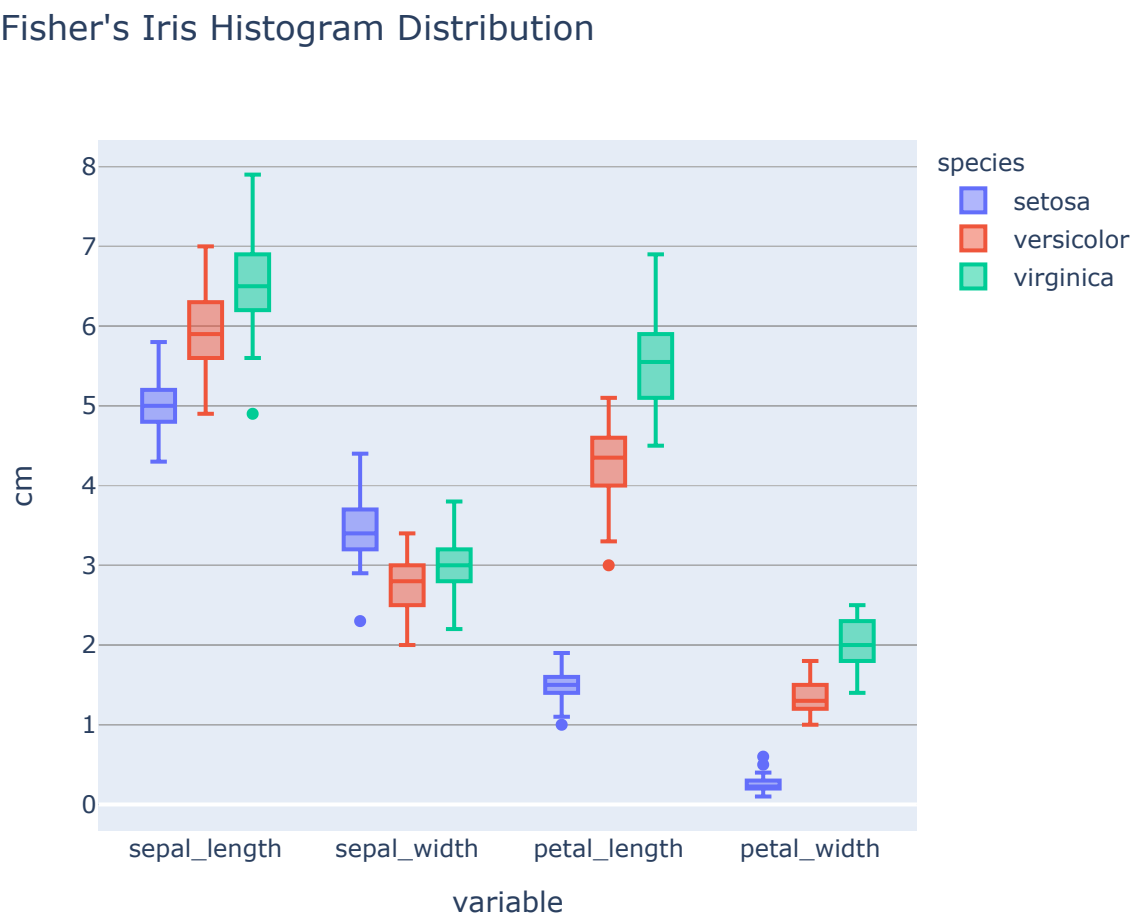
8) Use the pandas interface to generate a quick boxplot (df_iris.boxplot()!) However, look up how to created a faceted boxplot with each variable listed in a separate plot, automatically showing the distributions of your variables by "species".

In [11]:
```
fig = df_iris.boxplot(by="species", figsize=(10,8))
```



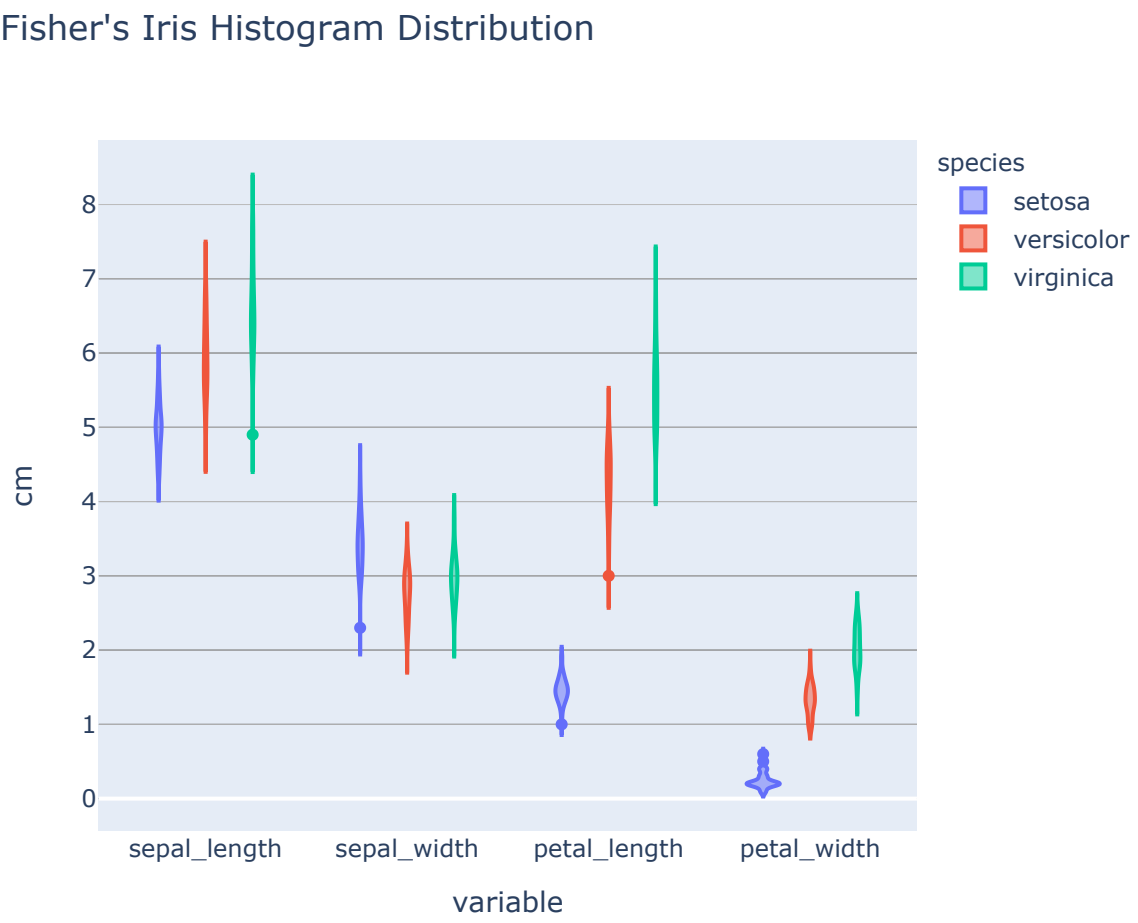
9) Use either seaborn or plotly to generate a boxplot over each variable, but now showing the three different species as distinct boxplots.

In [12]:
```
px.box(df_iris, color="species", title="Fisher's Iris Histogram Distribu
tion", labels={"value": "cm"})
```



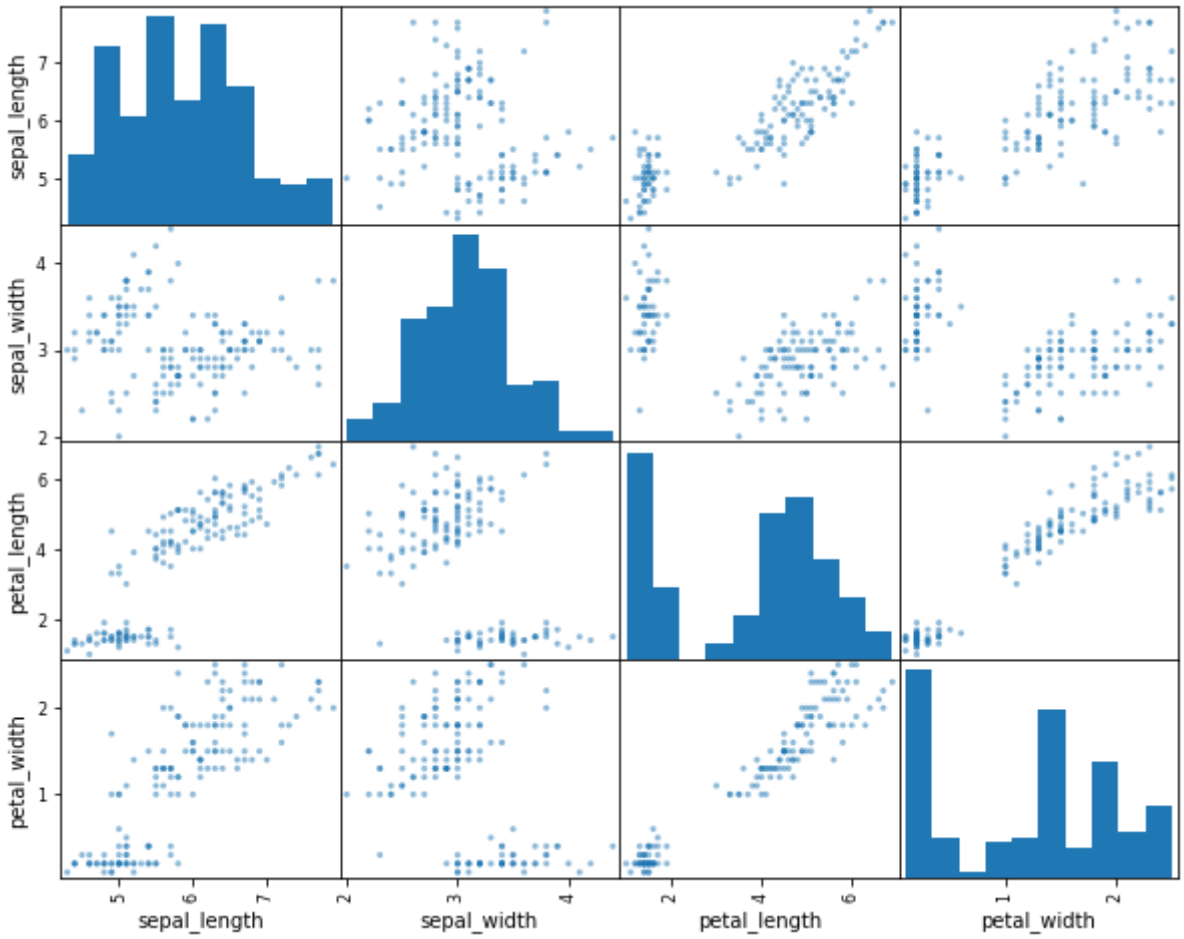
10) Generate a violin plot over all variables much like the previous boxplot, but again, be sure to indicate the species as a distinct color.

In [13]:
```
px.violin(df_iris, color="species", title="Fisher's Iris Histogram Distribution", labels={"value": "cm"})
```



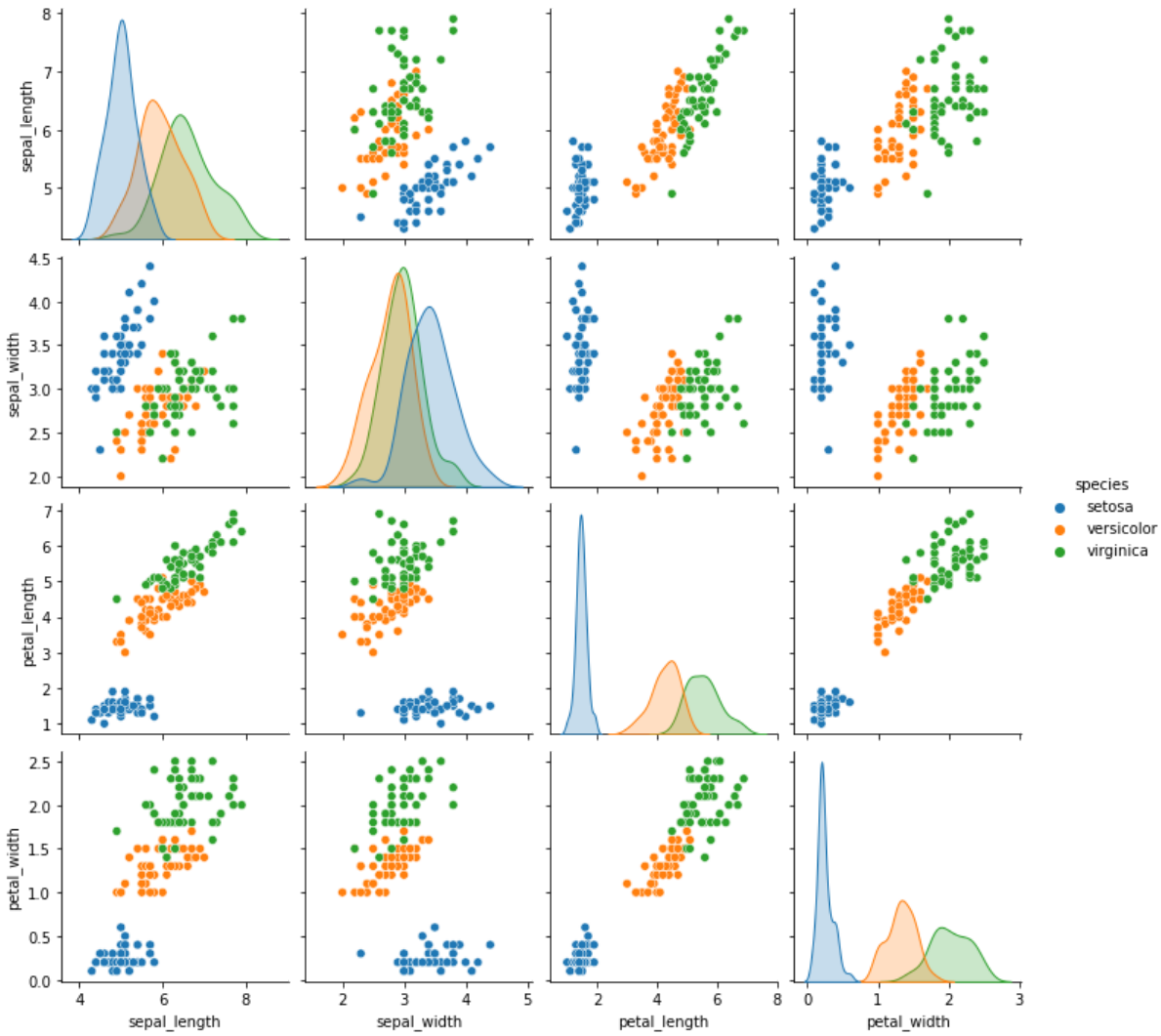
11) Read about the scatter_matrix() function in pandas. Use it to generate a scatterplot matrix, and use species for the color.

In [14]:
```
fig = pd.plotting.scatter_matrix(df_iris, figsize=(10,8))
```



12) Read about the seaborn pairplot() function, then use it to generate one of the most useful scatterplots matrices you'll see with this data. (It won't be too different than the scatter_matrix function, just easier to create, and the diagonal density plots are much better.) Take a moment to study the plot, and really try to understand just how much information this plot is conveying. Be sure to figure out how to distinguish the species by color.

In [15]:
```
fig = sns.pairplot(df_iris, hue="species")
```



13) From your observations, which species do you expect to have the best classifier performance? Why?

The best classifier for the species would be setosa, due to how much of a distinction it has from the other two in several comparisons. versicolor and virginica are more similar and will be more difficult to tell apart.

14) Split your data frame into X and y, where X represents only your four predictor variables, and y represents only the target class, species. Output the names of the columns and the shape of both just to confirm that they both have the same number of observations, and that the number of variables in each are correct. You should have (150, 4), and (150, 1) respectively

In [16]:
```
X = df_iris.loc[:, ["sepal_length","sepal_width","petal_length","petal_w
idth"]]
y = df_iris.loc[:, ["species"]]
```

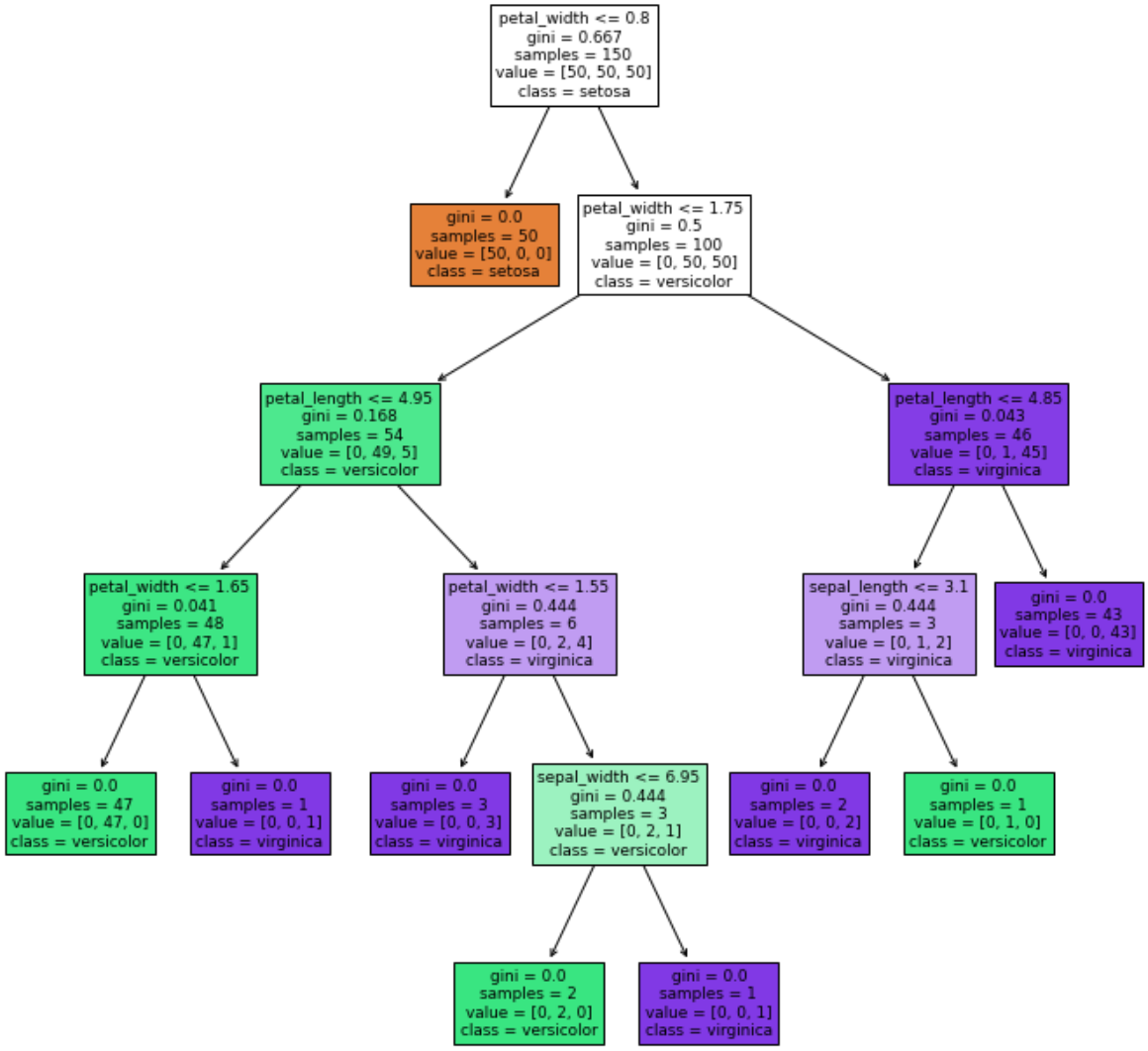
15) Create an instance of a decision tree classifier using DecisionTreeCassifier() with default parameters. Name the classifier clf. Train the classifier with the entire dataset (i.e. all of X and y.) Show the classifier after training by simply include print(clf)so you can see the default parameters used to build the classifier

In [17]:
```
from sklearn import tree

clf = tree.DecisionTreeClassifier()
clf.fit(X, y)
print(clf)
```

```
DecisionTreeClassifier()
```

16) Remember that one of the most popular reasons for using decision trees is because the model is easily visualized for model interpretation purposes. Use the plot_tree method to plot the tree. Explore the arguments to be sure that nodes are shaded by target class. Feature and class names should be shown. Your tree should look something like the following:

In [18]:
```
plt.figure(figsize=(12,12))
y_str = df_iris.species.unique()
x_str = ["sepal_width","sepal_length","petal_length","petal_width"]

fig = tree.plot_tree(clf, class_names=y_str, feature_names=x_str, filled
=True)
```



17) - Use this model to predict back your training data to evaluate your model. Name your predictions y_pred. Then, report the accuracy using the score method on the classifier.

In [19]:
```
y_pred = clf.predict(X)
clf.score(X, y)
```

Out[19]: 1.0

18) You should see 100% accuracy. Why are you getting a perfect score?

We are getting a perfect score because we are doing the worst mistake in machine learning: using test data from the training data. It is guaranteed to succeed because that is the info we used for the classification model itself.

19) Let's simplify our tree structure. Create a new tree, but adjust the pruning / complexity parameters. How? We'll keep this simple. Ensure that every leaf in the tree contains at least 5 samples. Show the tree that you induced, and again store your predictions as y_pred. Then, show the accuracy. (It should be 97.3%)

In [20]:
```
from sklearn.metrics import accuracy_score
plt.figure(figsize=(12,12))

clf = tree.DecisionTreeClassifier(min_samples_leaf=5)
clf.fit(X, y)
fig = tree.plot_tree(clf, class_names=y_str, feature_names=x_str, filled
=True)

y_pred = clf.predict(X)
clf.score(X, y)
```

Out[20]: 0.9733333333333334

20) OK – clearly we have lower accuracy. Dive deeper. Accuracy is usually not a good measure of classifier performance. Look up the function classification_report. This outputs a lot of performance information!

In [21]:
```
from sklearn import metrics

print(metrics.classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.98      0.94      0.96        50
   virginica       0.94      0.98      0.96        50

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

The lowerest precision is virginica, while the lowest recall is versicolor.

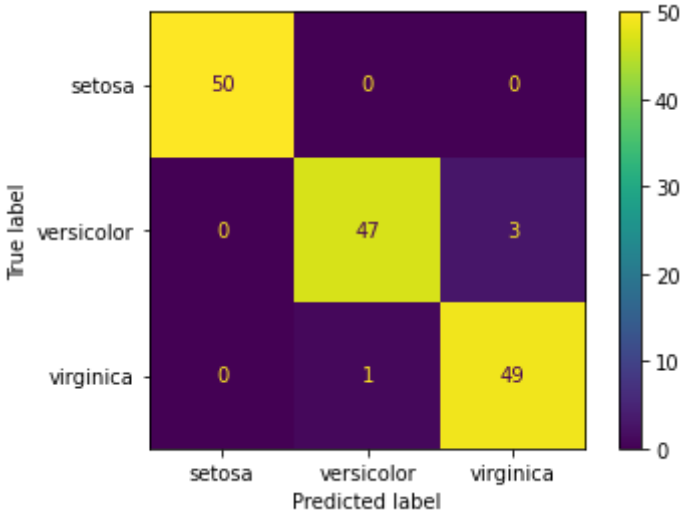
21) Output a confusion matrix using the confusion_matrix method in sklearn.metrics. Your result should look like a square matrix, where rows are the true labels, and the columns are the predicted labels, and the diagonal represents the cases where the true label and predicted label match.

In [22]:
```
metrics.confusion_matrix(y, y_pred)
```

Out[22]:
```
array([[50,  0,  0],
       [ 0, 47,  3],
       [ 0,  1, 49]])
```

22) Even better! Use the plot_confusion_matrix to output an excellent visual summary of the classifier performance. Your result should look as follows:

In [23]:
```
fig = metrics.plot_confusion_matrix(clf, X, y)
```



23) Interpret your confusion matrix and classification report. Which class is performing the best? Which is performing the worst? How many total incorrect predictions?

There are 4 incorrect predictions from the model that was generated. Setosa has no bad predictions, while there is error finding the difference between some versicolors and virginicas.
Setosa is best, versicolors is worst.

24) You need to create a train / test split of your data to properly validate your model. Read about the function called train_test_split. in the package sklearn.model_selection. Then, use it to split your data into an 70% / 30% split of training and testing data, respectively. You should end up with four data frames, denoted X_train, X_test, Y_train, Y_test. Use an initial random seed of 0. Be sure to shuffle the data (verify that this is a default setting.) Show the dimensions of each of these (i.e. how many entries in each?)

In [24]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.3
, random_state=0)
```

25) Create a new instance of DecisionTreeClassifier with an initial random seed value of 100, and a minimum number of samples in a leaf set to 5. Store the classifier as clf. Train your classifier with your training data.

In [25]:
```
clf = tree.DecisionTreeClassifier(min_samples_leaf=5, random_state=100)
clf.fit(X_train, y_train)
```

Out[25]:
```
DecisionTreeClassifier(min_samples_leaf=5, random_state=100)
```

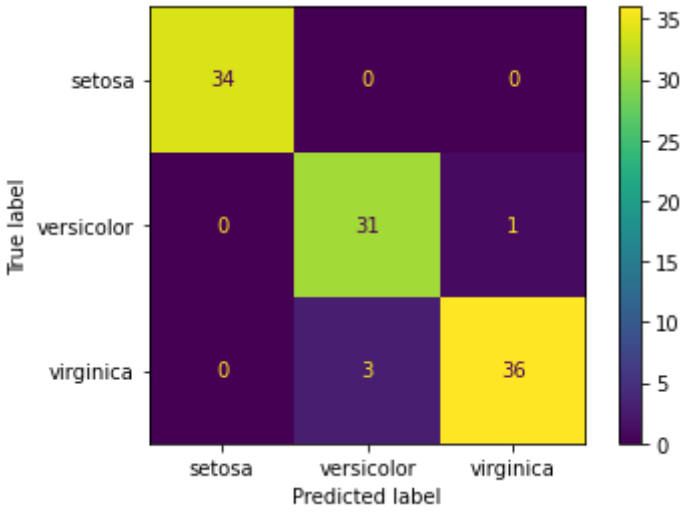
26) Use this model to predict the labels on your training data and your test data. Call your predictions y_pred_train. and y_pred_test. Show the accuracy of your classifier on both your training data and test data.

In [26]:
```
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)

acc_train = metrics.accuracy_score(y_train, y_pred_train)
acc_test = metrics.accuracy_score(y_test, y_pred_test)

print("Training accuracy:", str(round(acc_train*100, 2)) + '%')
print("Testing accuracy:", str(round(acc_test*100, 2)) + '%')
```

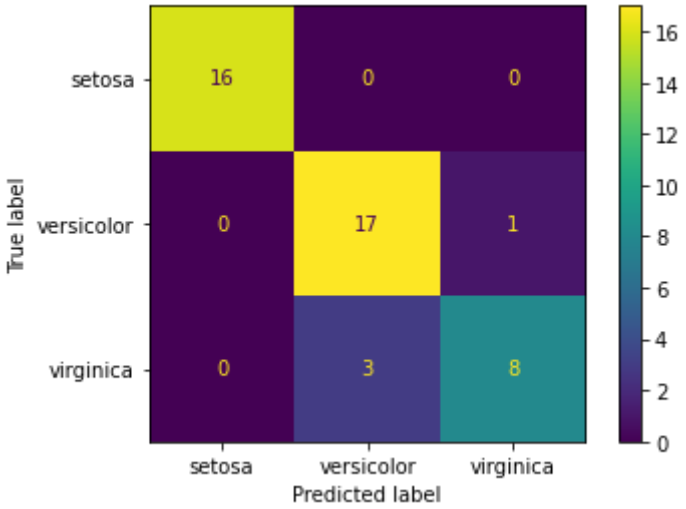
```
Training accuracy: 96.19%
Testing accuracy: 91.11%
```

27) Use the classification report and confusion matrix techniques discussed above to assess the performance of your classifier on both the training and the test data. Summarize your findings.

In [27]:
```
fig = metrics.plot_confusion_matrix(clf, X_train, y_train)
```



In [28]:
```
fig = metrics.plot_confusion_matrix(clf, X_test, y_test)
```



We can see from the two matrixes above that there are 4 bad predictions for each of the two cases. Althought 4 out of all of the tests is a bigger precent of bad predictions than 4 of the trainings. Again both cases show that it is a distinction between versicolors and virginica that are not correctly predicted.

28) Quite often, when we have misclassifications, it's important to take the time to dig into your test data to determine which observations are being misclassified. Use data selection techniques to output the data that are being misclassified in the test data only.

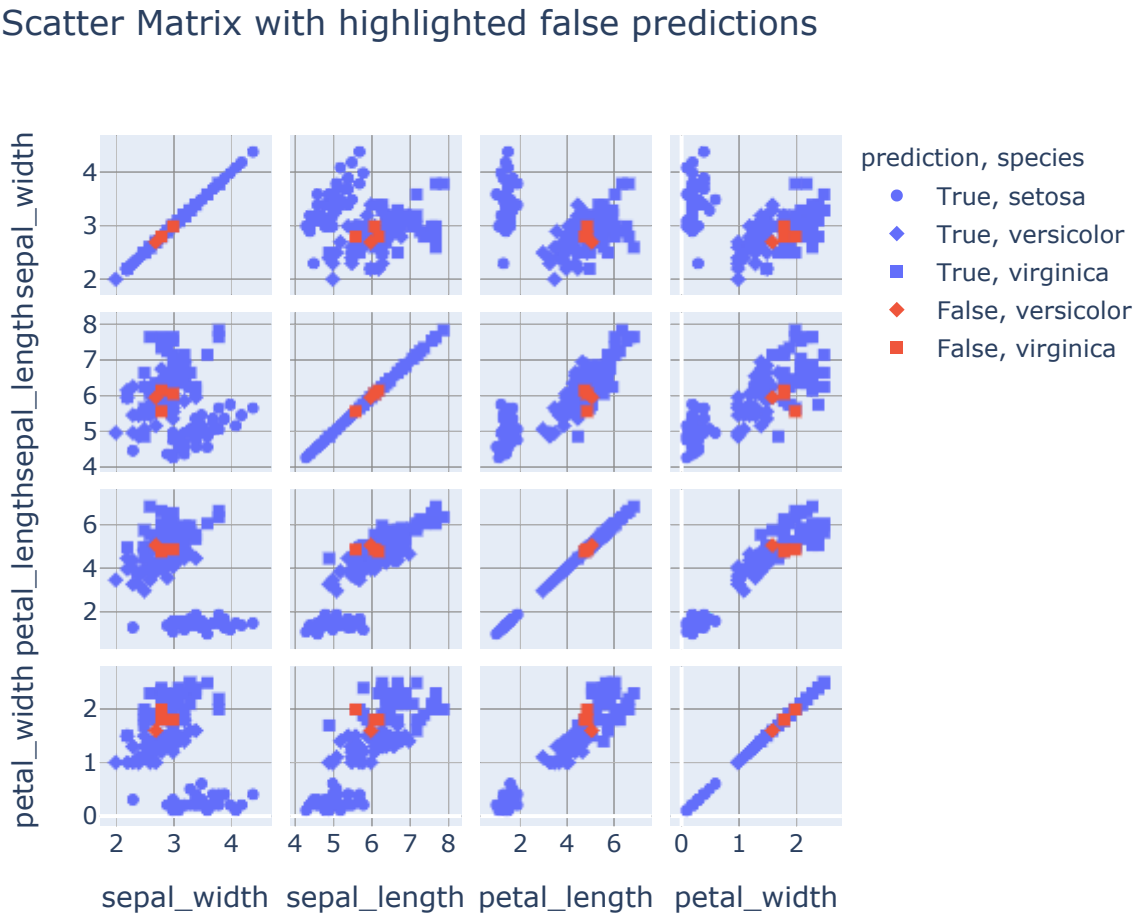
In [29]:
```
import time

t = time.time()
y_pred = y_test.copy()
y_pred.species = y_pred_test
mask = np.logical_not(np.equal(y_test, y_pred))
indices = mask[mask].dropna().index

X_test.loc[indices,:]
```

Out[29]:

	sepal_length	sepal_width	petal_length	petal_width
121	5.6	2.8	4.9	2.0
126	6.2	2.8	4.8	1.8
127	6.1	3.0	4.9	1.8
83	6.0	2.7	5.1	1.6

29) It's even more interesting when you can visualize where in your feature space your classifier might be missing something. Create a scatterplot matrix, but this time, highlight the instance(s) that are being misclassified.

In [30]:
```
df_iris_false = df_iris.copy()
df_iris_false["prediction"] = np.logical_not(mask)
df_iris_false.prediction.fillna(True, inplace=True)

px.scatter_matrix(df_iris_false, color="prediction", symbol="species", d
imensions=x_str, title="Scatter Matrix with highlighted false prediction
s")
```



30) Read about the KFold class. State what a KFold cross validation object will do for you. Then, create an instance of KFold with 10 splits, an initial seed of 100, and be sure to shuffle your data. Call your instance kfold.

In [31]:
```
from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=100)
```

kfold will help since it allows for us to use different permutations of sets as training vs testing(validation set). This makes it a more robust way of getting reliable models.

31) Read about the split() method for KFold

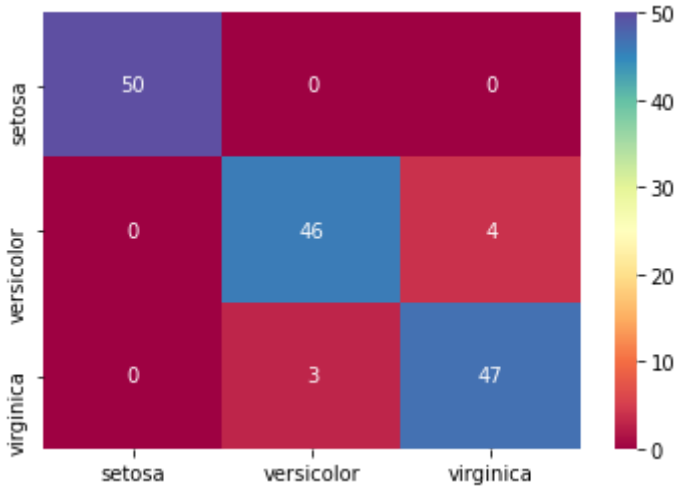
In [32]:
```
df_results = pd.DataFrame()
acc_score = []
pred = []

# clf = tree.DecisionTreeClassifier()
for train_index, test_index in kfold.split(X):
    X_train, X_test = X.iloc[train_index,:], X.iloc[test_index,:]
    y_train, y_test = y.iloc[train_index,:], y.iloc[test_index,:]

    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    pred.extend(list(zip(y_test.species, y_pred, y_test.index)))
    acc_score.append(accuracy_score(y_pred, y_test))

df_results = pd.DataFrame(pred, columns=["true", "predicted", "indexing"
])
df_results.set_index("indexing", inplace=True)
acc_score
```

Out[32]:
```
[1.0,
 0.9333333333333333,
 1.0,
 1.0,
 0.8666666666666667,
 0.8,
 0.9333333333333333,
 1.0,
 1.0,
 1.0]
```

32) Print out a classification report from your 10 fold cross validation. Also print out your confusion matrix. NOTE: You won't be able to use the plot_confusion_matrix method here, but you should be able to use the confusion_matrix method and seaborn's heatmap method. For example (using a poor color map!):

In [33]:
```
print(metrics.classification_report(df_results.true, df_results.predicted))

cm = metrics.confusion_matrix(df_results.true, df_results.predicted)
heat_map = sns.heatmap(cm, cmap="Spectral", annot=True, xticklabels=y_str, yticklabels=y_str)
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.94      0.92      0.93        50
   virginica       0.92      0.94      0.93        50

    accuracy                           0.95       150
   macro avg       0.95      0.95      0.95       150
weighted avg       0.95      0.95      0.95       150
```



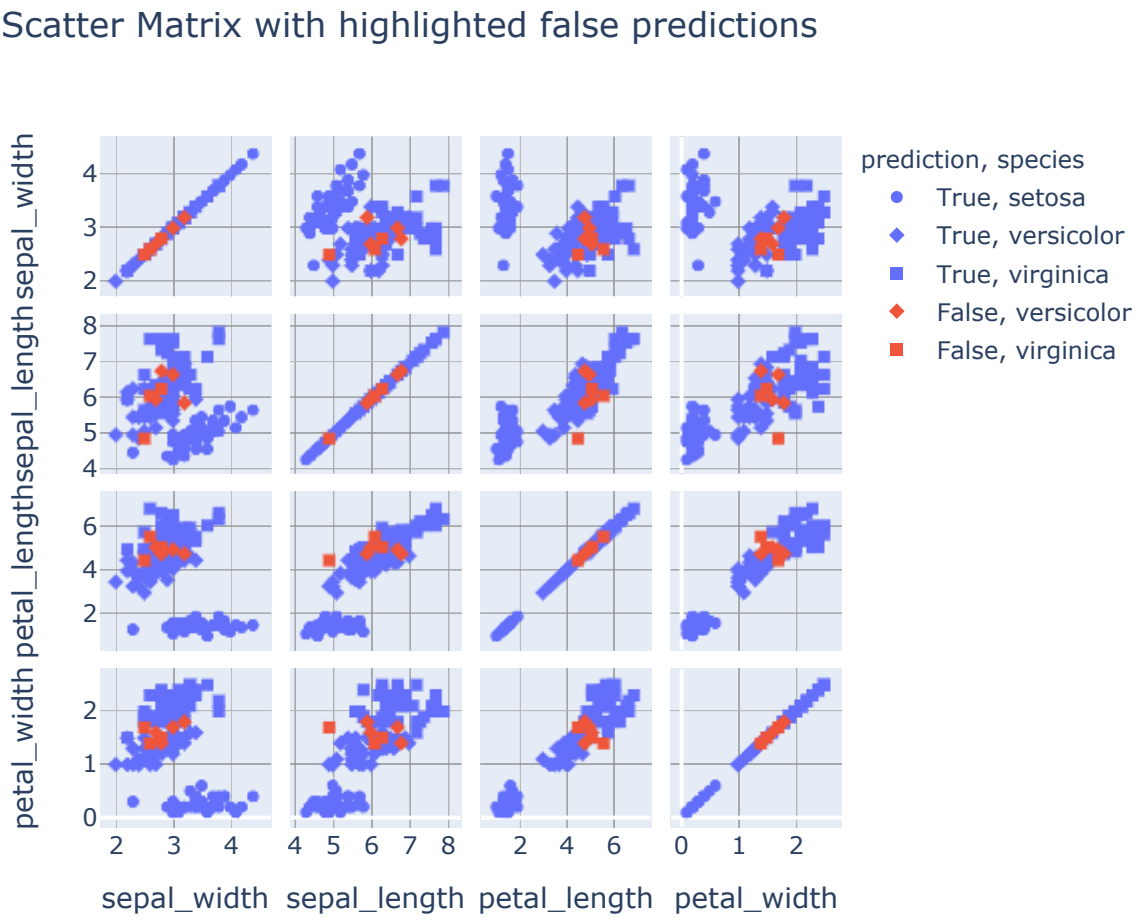
33) Select and generate a report all of your test instances that were misclassified from the 10 fold cross validation. Also, generate a scatterplot that highlights the location of the instances that were misclassified.

In [34]:
```
misclassified = np.equal(df_results.true, df_results.predicted)
df_results.loc[misclassified[~misclassified].index]
```

Out[34]:

indexing	true	predicted
77	versicolor	virginica
133	virginica	versicolor
134	virginica	versicolor
70	versicolor	virginica
76	versicolor	virginica
83	versicolor	virginica
106	virginica	versicolor

In [35]:
```
df_iris_false = df_iris.copy()
df_iris_false["prediction"] = misclassified
px.scatter_matrix(df_iris_false, color="prediction", symbol="species", d
imensions=x_str, title="Scatter Matrix with highlighted false prediction
s")
```



34) As you would expect, the scikit-learn framework has some powerful methods that can run an entire cross validation and report whatever metrics you want. Read about the cross_validate method, then use it to run a 10-fold cross validation on a default decision tree, reporting back 'accuracy' and 'f1_macro' measurements on both the training and testing data. Report your results as a single data frame.

In [36]:
```
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer, f1_score

scores = {'accuracy':'accuracy', 'f1_macro':make_scorer(f1_score, averag
e='macro')}
c = cross_validate(tree.DecisionTreeClassifier(), X, y, cv=10, scoring=s
cores, return_train_score=True)
pd.DataFrame(c)
```

Out[36]:

	fit_time	score_time	test_accuracy	train_accuracy	test_f1_macro	train_f1_macro
0	0.003687	0.002229	1.000000	1.0	1.000000	1.0
1	0.003873	0.002040	0.933333	1.0	0.932660	1.0
2	0.002969	0.002198	1.000000	1.0	1.000000	1.0
3	0.003438	0.002082	0.933333	1.0	0.932660	1.0
4	0.003292	0.002430	0.933333	1.0	0.932660	1.0
5	0.002960	0.001942	0.866667	1.0	0.866667	1.0
6	0.003293	0.002510	0.933333	1.0	0.932660	1.0
7	0.002778	0.002244	1.000000	1.0	1.000000	1.0
8	0.002591	0.001813	1.000000	1.0	1.000000	1.0
9	0.002657	0.001856	1.000000	1.0	1.000000	1.0

35) On the above, what are the variables fit_time and score_time?

fit time is the time it takes to call the fit function that is implemented by the estimator parameter passed in.
score time is the time that it takes to call the predict method.

36) OK, one last function for validation purposes. Read about the function cross_val_predict. This is perhaps among the most powerful of the model selection functions provided by sklearn, as it will generate predictions. You can then use these predictions to run a classification_report and report confusion matrices. Use the cross_val_predict function to run a 10-fold cross validation with a default decision tree, and print the classification_report on your results.#%%

In [37]:
```
from sklearn.model_selection import cross_val_predict

y_pred = cross_val_predict(tree.DecisionTreeClassifier(), X, y, cv=10)
print(metrics.classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.94      0.94      0.94        50
   virginica       0.94      0.94      0.94        50

    accuracy                           0.96       150
   macro avg       0.96      0.96      0.96       150
weighted avg       0.96      0.96      0.96       150
```

37) For your last task, you will perform model comparison tasks. Use the cross_val_predict method to compare the predictive performance on the following models: a. A default decision tree b. A decision tree with "entropy" for measuring impurity c. A KNeighborsClassifier with a two different values of k (.) d. MultinomialNB classifier (sklearn's Naïve Bayes implementation) Compare and contrast the performance results between the different models. Which one would you choose?

In [38]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import MultinomialNB
from time import time
clfs = [tree.DecisionTreeClassifier(),
        tree.DecisionTreeClassifier(criterion="entropy"),
        KNeighborsClassifier(n_neighbors=1),
        KNeighborsClassifier(n_neighbors=5),
        MultinomialNB()]

for clf in clfs:
    init = time()
    y_pred = cross_val_predict(clf, X, np.ravel(y), cv=10)
    report = metrics.classification_report(y, y_pred)
    reportd = metrics.classification_report(y, y_pred, output_dict=True)

    print(type(clf).__name__)
    print("time:", round((time() - init) * 1000, 3) , 'ms')
    print("accuracy:", str(round(reportd['accuracy'] * 100, 2)) + '%')
    print(report)
    print("\n")
```

```
DecisionTreeClassifier
time: 33.57 ms
accuracy: 95.33%
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.92      0.94      0.93        50
   virginica       0.94      0.92      0.93        50

    accuracy                           0.95       150
   macro avg       0.95      0.95      0.95       150
weighted avg       0.95      0.95      0.95       150



DecisionTreeClassifier
time: 38.81 ms
accuracy: 96.0%
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.94      0.94      0.94        50
   virginica       0.94      0.94      0.94        50

    accuracy                           0.96       150
   macro avg       0.96      0.96      0.96       150
weighted avg       0.96      0.96      0.96       150



KNeighborsClassifier
time: 47.83 ms
accuracy: 96.0%
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.94      0.94      0.94        50
   virginica       0.94      0.94      0.94        50

    accuracy                           0.96       150
   macro avg       0.96      0.96      0.96       150
weighted avg       0.96      0.96      0.96       150



KNeighborsClassifier
time: 56.658 ms
accuracy: 96.67%
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.98      0.92      0.95        50
   virginica       0.92      0.98      0.95        50

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

```
MultinomialNB
time: 46.257 ms
accuracy: 95.33%
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.94      0.92      0.93        50
   virginica       0.92      0.94      0.93        50

    accuracy                           0.95       150
   macro avg       0.95      0.95      0.95       150
weighted avg       0.95      0.95      0.95       150
```

The highest accuracy all around is the KNN classifier, but the trade off is that the time to calculate this is significantly higher, especially as the training set grows.
It also seems like if we choose the DTree route, it would seem that this specific set has no difference between gini and entropy. The naive bayes implementation seems to be fine too but not the best out of all.

So for this smaller dataset the k=5 Neighborhood Classifier would be best suited.