Simple PCA Example

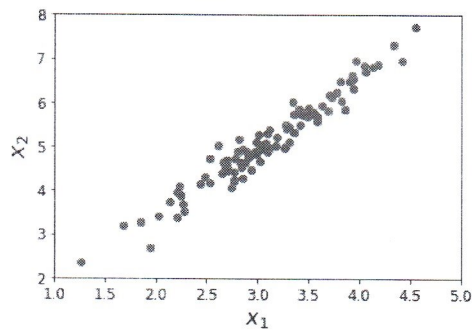
This is a simple example to show how PCA works and what the principal components represent. First we load the libraries we need. Next we generate some data, we normalize them using the standard scaler, we apply the PCA and plot the components over the normalized data.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.datasets import make_blobs
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

plt.rcParams["font.family"] = 'sans'
plt.rcParams["font.size"] = 12
plt.rcParams["axes.titlesize"] = 24
plt.rcParams["axes.labelsize"] = 18
```

In [2]:
```
def DrawVector(v0, v1, ax=None):
    ax = ax or plt.gca()
    ax.annotate('', v1, v0, arrowprops={'arrowstyle': '->', 'linewidth': 2, 'shrinkA': 0, 'shrinkB': 0})
```

In [3]:
```
n_samples = 100
random_state=1234
np.random.seed(random_state)
```

In [4]:
```
X, y = make_blobs(n_samples=n_samples, centers=1, random_state=random_state,center_box=(2.0, 5.0))
X = np.dot(X,[[0.6, .8], [0.4, .8]])
plt.scatter(X[:,0],X[:,1]);
x_lim = [np.floor(np.min(X[:,0])),np.ceil(np.max(X[:,0]))]
y_lim = [np.floor(np.min(X[:,1])),np.ceil(np.max(X[:,1]))]
plt.xlim(x_lim)
plt.ylim(y_lim)
plt.xlabel("$x_1$");
plt.ylabel("$x_2$");
```

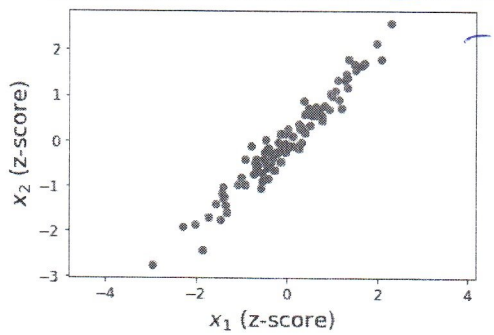


In [5]:
```
Xs = StandardScaler().fit_transform(X)
plt.scatter(Xs[:,0],Xs[:,1]);
xs_lim = [np.floor(np.min(Xs[:,0])),np.ceil(np.max(Xs[:,0]))]
ys_lim = [np.floor(np.min(Xs[:,1])),np.ceil(np.max(Xs[:,1]))]
plt.xlim(xs_lim)
plt.ylim(ys_lim)
plt.xlabel("$x_1$ (z-score)");
plt.ylabel("$x_2$ (z-score)");
plt.axis('equal');
```

→ if we want to apply PCA we have to standardize data ($(data - \mu)/\sigma$)

This command does it all: it evaluates μ and σ and it standardizes data

→ it's used to not have deformation scale along the axes



— almost the same shape

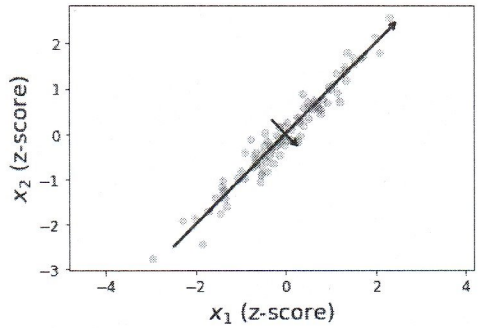
In [6]:
```
pca = PCA(n_components=2)
pca.fit(Xs)
```

— we're doing a PCA with the same number of components as the number of features ⇒ we're looking for the optimal data representation (in the commands we first create the PCA with 2 components and then we apply it)

Out[6]:
```
PCA(n_components=2)
```

In [7]:
```
plt.scatter(Xs[:,0],Xs[:,1],alpha=0.2);
xs_lim = [np.floor(np.min(Xs[:,0])),np.ceil(np.max(Xs[:,0]))]
ys_lim = [np.floor(np.min(Xs[:,1])),np.ceil(np.max(Xs[:,1]))]
plt.xlabel("$x_1$ (z-score)");
plt.ylabel("$x_2$ (z-score)");

for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 2.5 * np.sqrt(length)
    plt.annotate('', pca.mean_-v, pca.mean_+v, arrowprops={'arrowstyle': '->', 'linewidth': 2, 'shrinkA': 0, 'shrinkB': 0})
plt.axis('equal');
```

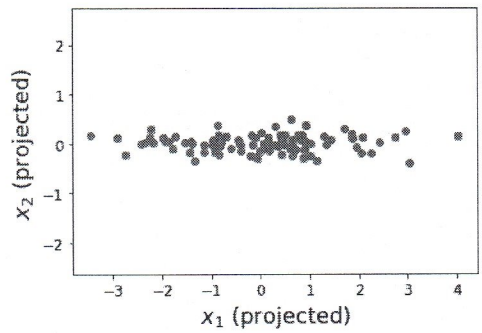
How we plot the principal components



If we now project the original data over the two components, we note that the projected data are aligned with respect to the two axis.

In [8]:
```
Xs_pca = pca.transform(Xs)
```
→ we transform the data in the Principal components space

In [9]:
```
plt.scatter(Xs_pca[:,0],Xs_pca[:,1]);
xs_lim = [np.floor(np.min(Xs[:,0])),np.ceil(np.max(Xs[:,0]))]
ys_lim = [np.floor(np.min(Xs[:,1])),np.ceil(np.max(Xs[:,1]))]
plt.xlim(xs_lim)
plt.ylim(ys_lim)
plt.xlabel("$x_1$ (projected)");
plt.ylabel("$x_2$ (projected)");
plt.axis('equal');
```



In [ ]:

Data Exploration - Iris Dataset

In this notebook we perform basic data exploration on the Iris data set:

https://en.wikipedia.org/wiki/Iris_flower_data_set

First, we load the libraries we need for the analysis.

In [1]:
```
# dataframe management
import pandas as pd

# numerical computation
import numpy as np

# visualization library
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.set_context(rc={"font.family":'sans',"font.size":24,"axes.titlesize":24,"axes.labelsize":24})


# import matplotlib and allow it to plot inline
import matplotlib.pyplot as plt
%matplotlib inline

# seaborn can generate several warnings, we ignore them
import warnings
warnings.filterwarnings("ignore")

# import the dataset library
from sklearn import datasets
```

Let's load the Iris dataset

In [2]:
```
dataset = datasets.load_iris()
```

In [3]:
```
print(dataset.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

In [4]:
```
# create data with input values
iris = pd.DataFrame(dataset.data, columns=dataset.feature_names)
```

In [5]:
```
# create target variable
iris["Species"] = dataset.target_names[dataset.target]
```

In [8]:
```
target_variable = 'Species'
input_variables = iris.columns[iris.columns!=target_variable]
```

Let's get some statistics for continuous attributes

In [9]:
```
iris.describe()
```

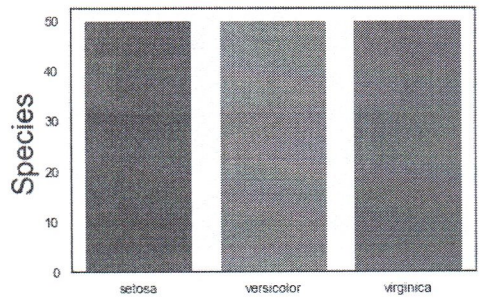
Out[9]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
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

And some statistics about the class attribute Species. We can use barplot to show the number of instances belonging to each class. As we can see, the dataset is completely balanced with 50 cases for each class.

In [11]:
```
sns.barplot(x=iris[target_variable].unique(),y=iris[target_variable].value_counts().sort_index());
```

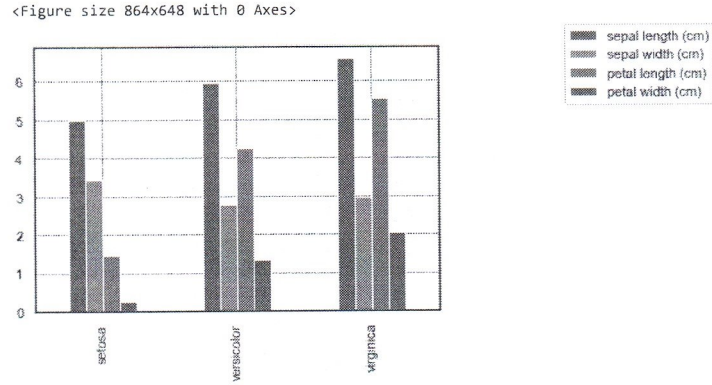
← barplot for nominal variable



We can use barplots also to plot summary statistics for each class value. For example, we can plot compute the mean values of each attribute.

In [26]:
```
plt.figure(figsize=(12, 9));
iris_gb=iris.groupby([target_variable]).mean();
iris_gb.plot(kind="bar");
plt.grid(color='black', linestyle='--', linewidth=.5);
plt.yticks(np.arange(0, 7, step=1.0));
plt.xlabel("");
plt.legend(loc='upper right',bbox_to_anchor=(1.7, 1.1));
```

```
<Figure size 864x648 with 0 Axes>
```



Let's plot the distribution of SepalLengthCm

In [29]:
```
from scipy.stats import iqr

numerical_variables = iris.columns[iris.columns!='Species']

print('Variable Range')
for c in numerical_variables:
    print('%s\t%.3f'%(c,np.max(iris[c]) - np.min(iris[c])))

print('\n\nInterquartile Range')
for c in numerical_variables:
    print('%s\t%.3f'%(c,iqr(iris[c])))
```

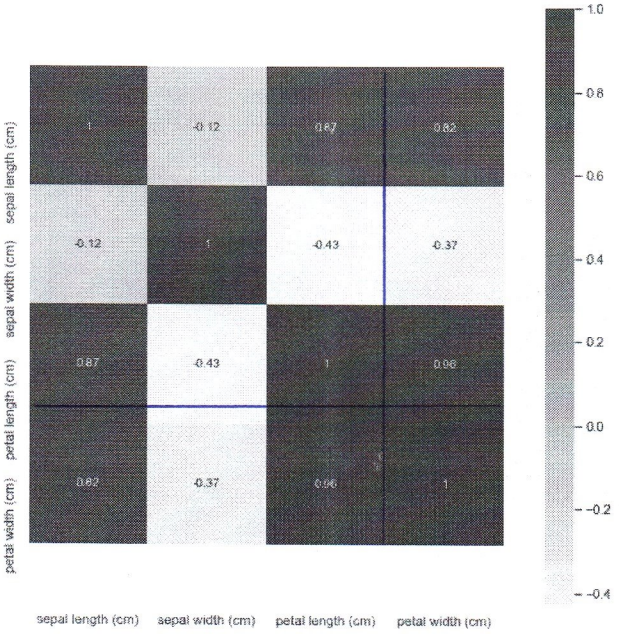
```
Variable Range
sepal length (cm)       3.600
sepal width (cm)        2.400
petal length (cm)       5.900
petal width (cm)        2.400


Interquartile Range
sepal length (cm)       1.300
sepal width (cm)        0.500
petal length (cm)       3.500
petal width (cm)        1.500
```

We can compute correlations among attributes.

In [30]:
```
corrmat = iris.corr()
plt.figure(figsize=(12,9))
sns.heatmap(corrmat, square=True, cmap="Blues",annot=True);
```

```
# these lines are here only to correct a matplotlib bug
b, t = plt.ylim() # discover the values for bottom and top
b += 0.5 # Add 0.5 to the bottom
t -= 0.5 # Subtract 0.5 from the top
plt.ylim(b, t) # update the ylim(bottom, top) values
#
```

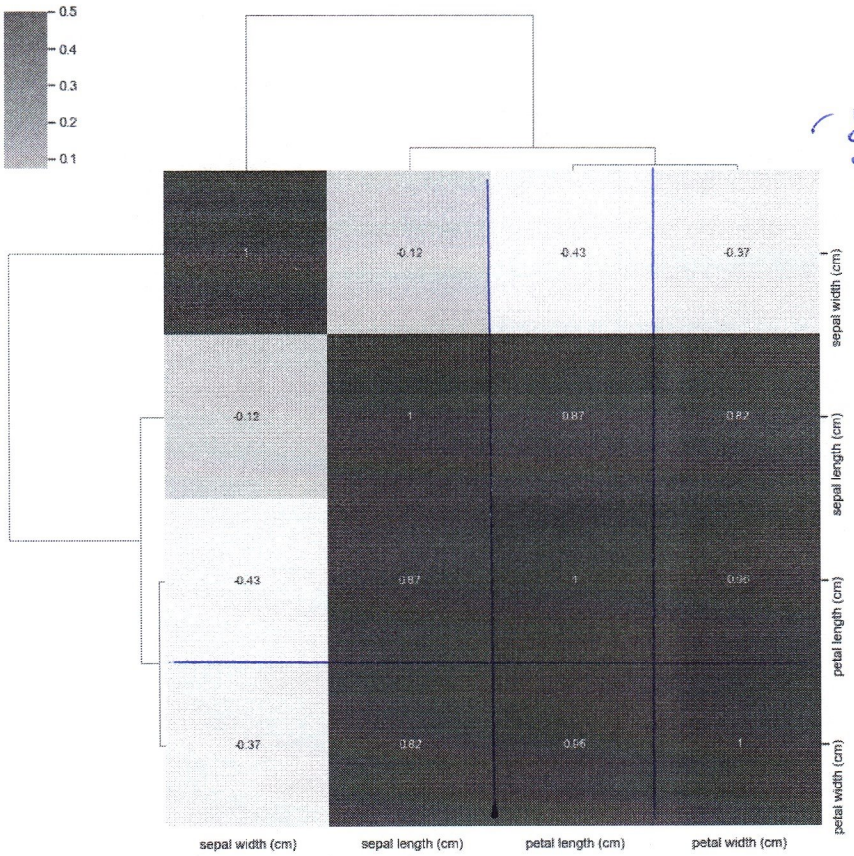
Out[30]: (4.5, -0.5)



We can further analyze the relations among variables by using clustermaps on the correlation matrix

In [31]:
```
plt.figure(figsize=(8,6))
sns.clustermap(iris.corr(), square=True, annot=True, cmap="Blues");
# these lines are here only to correct a matplotlib bug
b, t = plt.ylim() # discover the values for bottom and top
b += 0.5 # Add 0.5 to the bottom
t -= 0.5 # Subtract 0.5 from the top
plt.ylim(b, t); # update the ylim(bottom, top) values
#
```

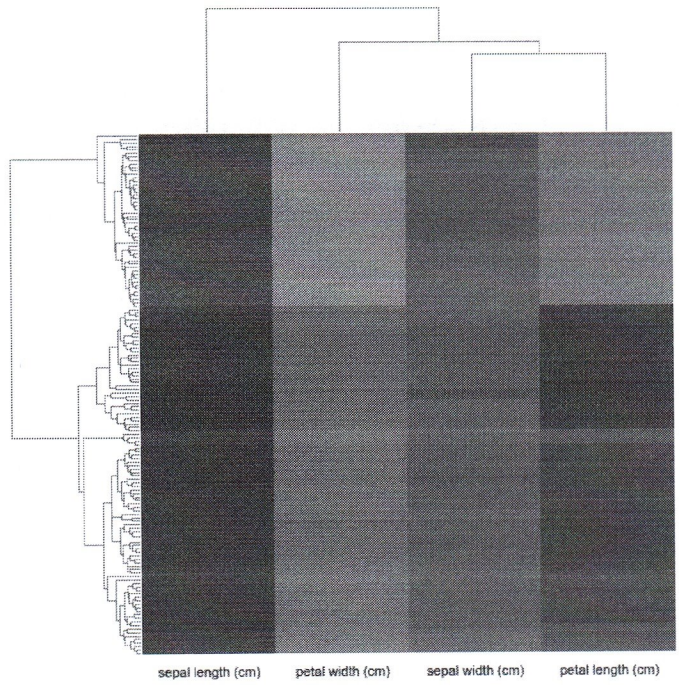
<Figure size 576x432 with 0 Axes>



we take the correlation and we cluster both columns and rows (there are two different clusters, however the matrix is symmetric so it's actually the same)

Or over the original dataset

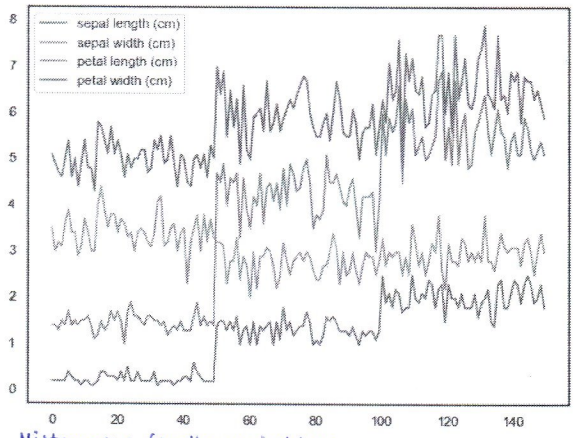
In [39]:
```
cm = sns.clustermap(iris[numerical_variables], center=0, cmap="Blues", figsize=(8, 8),yticklabels=False)
cm.cax.set_visible(False)
```



here we're clustering over all the dataset (two clusters: rows and columns)

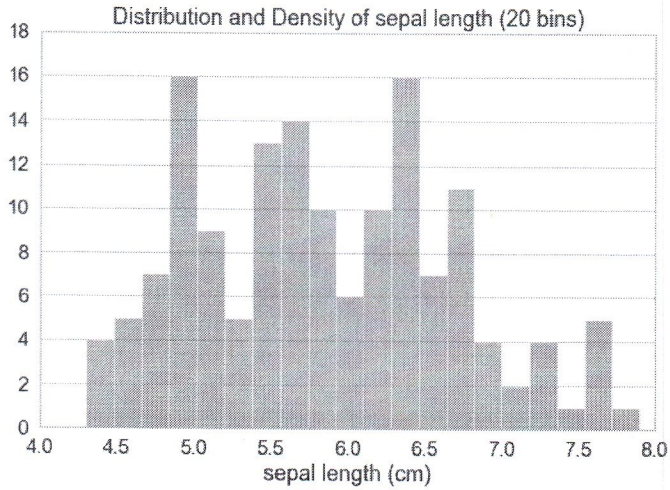
We check how features vary with each data input. The plot shows a sorting in the input values.

In [40]:
```
plt.figure(figsize=(8,6))
for feature in iris.columns[0:4]:
    plt.plot(iris[feature], label = feature)
plt.legend(loc='best');
```



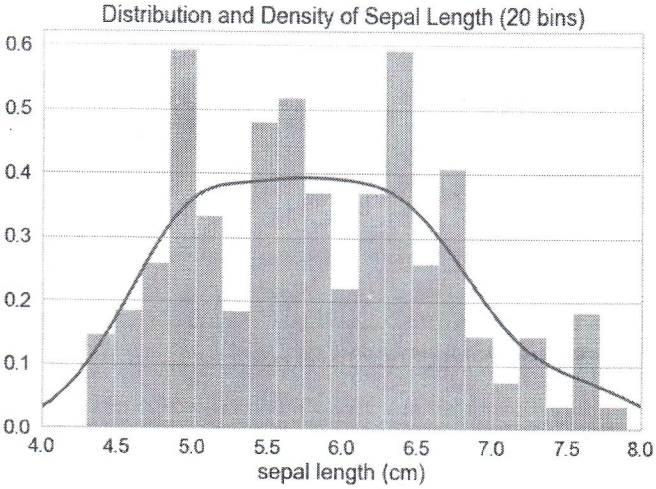
Histograms for the variables:

In [45]:
```
plt.figure(figsize=(8, 6))
sns.set_style("whitegrid")
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
dp = sns.distplot(iris['sepal length (cm)'],kde=False,bins=20)
dp.set_title('Distribution and Density of sepal length (20 bins)');
plt.tight_layout();
plt.grid(axis='x')
plt.xlim([4,8])
plt.yticks(np.arange(0, 20, step=2.0));
```

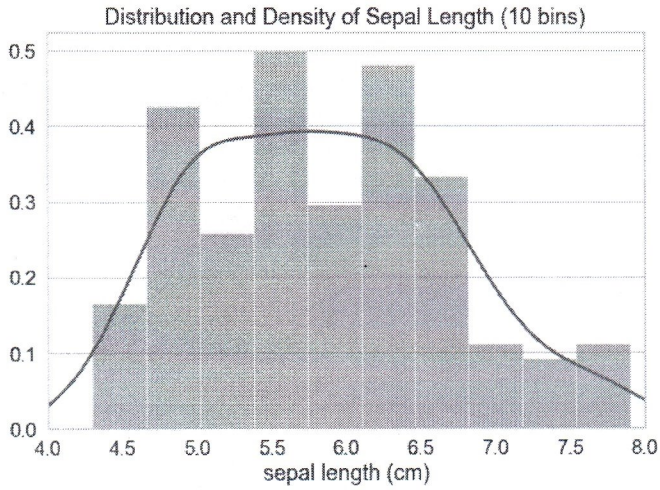


We can add the kernel density estimator to the plot, although it might not provide reliable information.

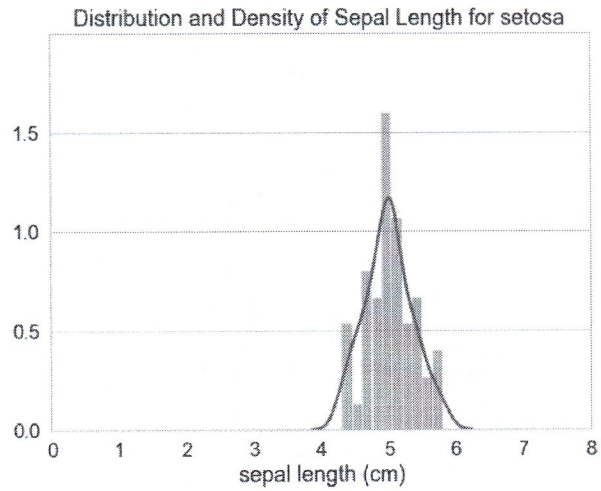
In [46]:
```
plt.figure(figsize=(8, 6))
sns.set_style("whitegrid")
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
dp = sns.distplot(iris['sepal length (cm)'],bins=20)
dp.set_title('Distribution and Density of Sepal Length (20 bins)');
plt.grid(axis='x')
plt.xlim([4,8])
plt.tight_layout();
```



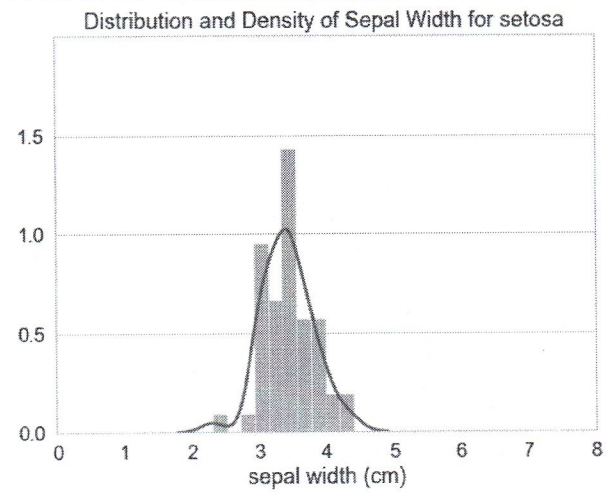
In [47]:
```
plt.figure(figsize=(8, 6))
sns.set_style("whitegrid")
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
dp = sns.distplot(iris['sepal length (cm)'],bins=10)
dp.set_title('Distribution and Density of Sepal Length (10 bins)');
plt.grid(axis='x')
plt.xlim([4,8])
plt.tight_layout();
```



In [48]:
```
plt.figure(figsize=(8, 6))
iris_is = iris[iris['Species'] == 'setosa']
hist1 = sns.distplot(iris_is['sepal length (cm)'],bins=10)
hist1.set_title('Distribution and Density of Sepal Length for setosa');
plt.xlim([0,8])
plt.ylim([0,2])
plt.yticks(np.arange(0,2,0.5))
plt.grid(axis='x')
```



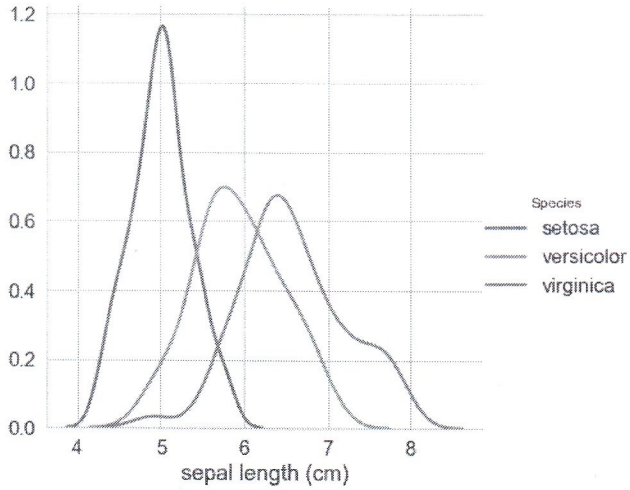
In [49]:
```
plt.figure(figsize=(8, 6))
iris_is = iris[iris['Species'] == 'setosa']
hist2 = sns.distplot(iris_is['sepal width (cm)'],bins=10)
hist2.set_title('Distribution and Density of Sepal Width for setosa');
plt.xlim([0,8])
plt.ylim([0,2])
plt.yticks(np.arange(0,2,0.5))
plt.grid(axis='x')
```



We can plot the distribution for each class.

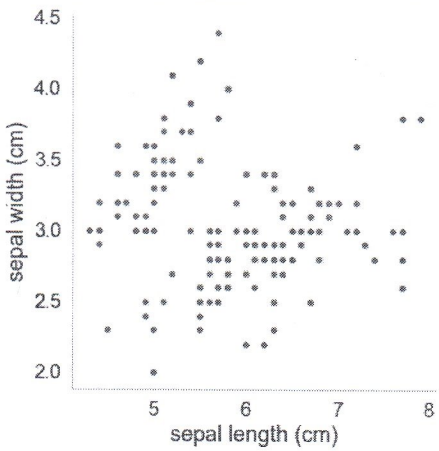
In [51]:
```
plt.figure(figsize=(8, 6))
sns.FacetGrid(iris, hue="Species", size=6) \
   .map(sns.kdeplot, "sepal length (cm)") \
   .add_legend();
```

```
<Figure size 576x432 with 0 Axes>
```



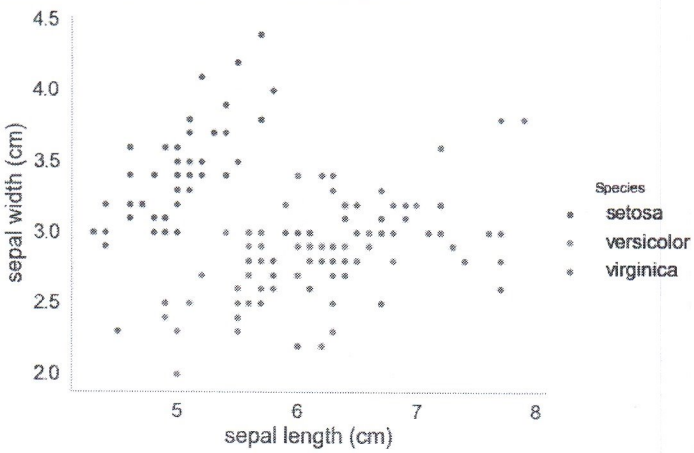
And now we use scatter plots.

In [52]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
sns.pairplot(iris, x_vars=["sepal length (cm)"], y_vars=["sepal width (cm)"], size=5).add_legend()
plt.grid(False)
```



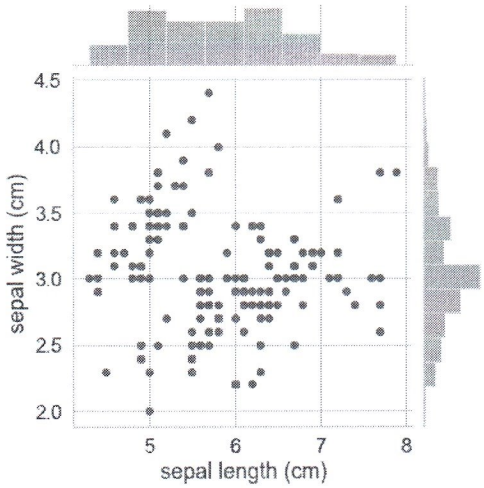
We can also add the information about the class.

In [53]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5})
sns.pairplot(iris, x_vars=["sepal length (cm)"], y_vars=["sepal width (cm)"], hue="Species", size=5).add_legend();
plt.grid(False)
```

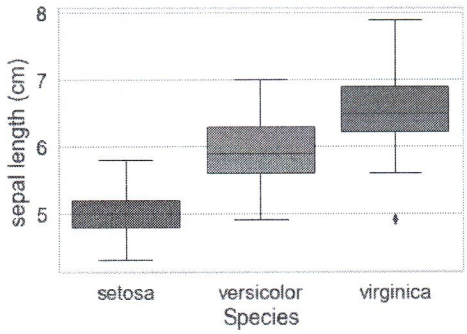


We can combine histograms and histograms in the same figure.

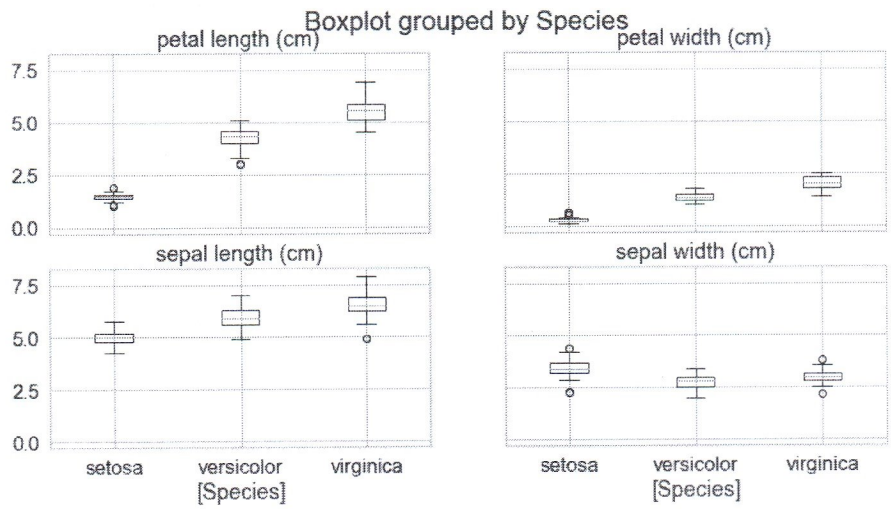
In [54]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 2.5});
sns.jointplot(x="sepal length (cm)", y="sepal width (cm)", data=iris);
plt.grid(False);
```



In [55]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 1})
sns.boxplot(x="Species", y="sepal length (cm)", data=iris);
```

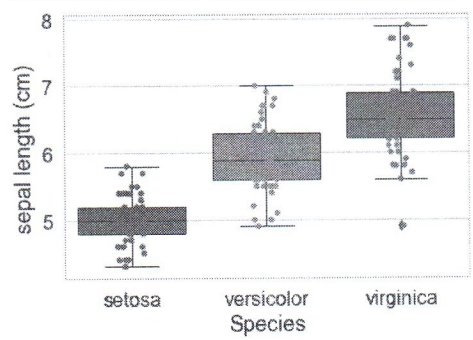


In [58]:
```
iris.boxplot(by="Species", figsize=(12, 6));
```

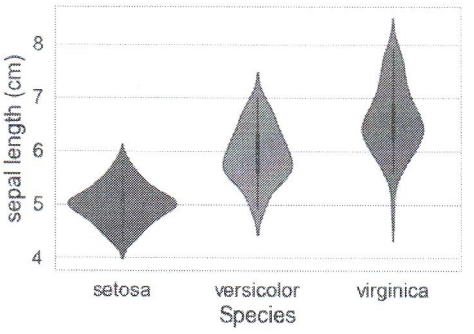


We can also add the scatter plot for every boxplot.

In [59]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 1})
ax = sns.boxplot(x="Species", y="sepal length (cm)", data=iris)
ax = sns.stripplot(x="Species", y="sepal length (cm)", data=iris, jitter=True, edgecolor="gray");
```

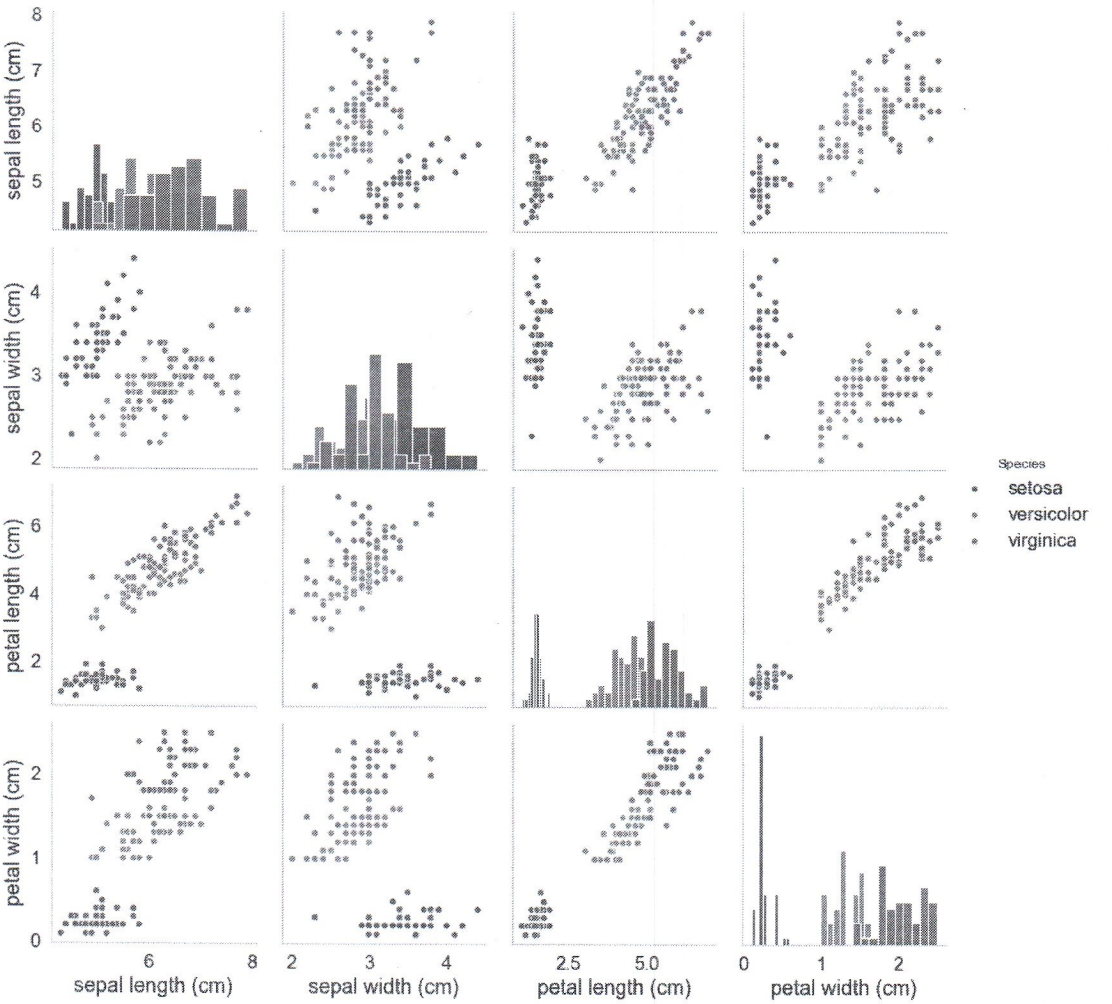


In [60]:
```
sns.set_context("notebook", font_scale=1.5, rc={"lines.linewidth": 1})
sns.violinplot(x="Species", y="sepal length (cm)", data=iris);
```



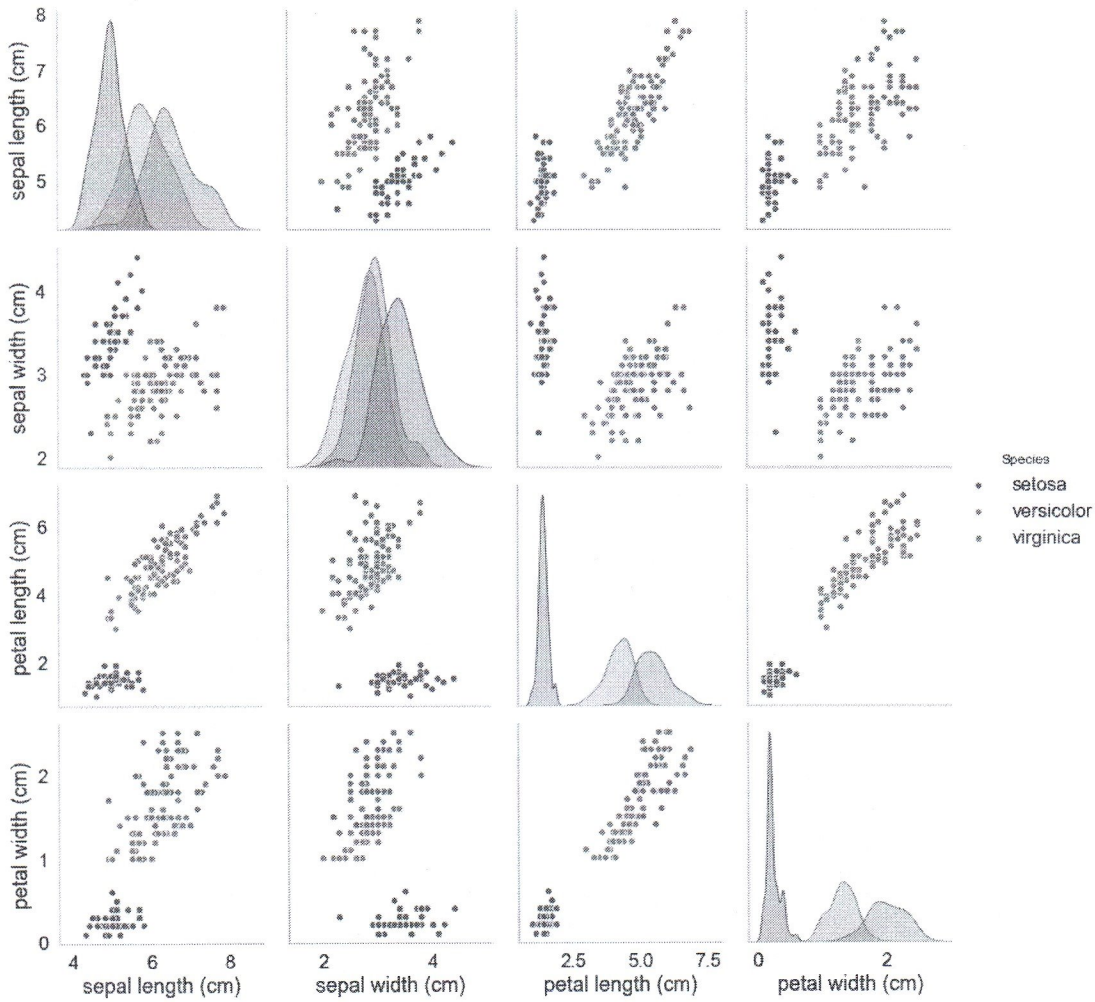
We can plot the scatter plots for the pairwise attribute combinations.

In [61]:
```
sns.set_style("whitegrid", {'axes.grid' : False})
sns.pairplot(iris, hue="Species", size=3, diag_kind="hist");
```



We can replace the diagonal bar plots with a gaussian kernel density estimate.

In [62]:
```
sns.set_style("whitegrid", {'axes.grid' : False})
sns.pairplot(iris, hue="Species", size=3, diag_kind="kde");
```



Principal Component Analysis

– we perform a PCA with 2 components so that we can plot it

So far we used only some data dimensions for visualization. We now apply Principal Component Analysis to project the four original dimensions into a two dimensional space.

In [63]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

In [64]:
```
x = iris.loc[:, numerical_variables].values
y = iris.loc[:,['Species']].values
```

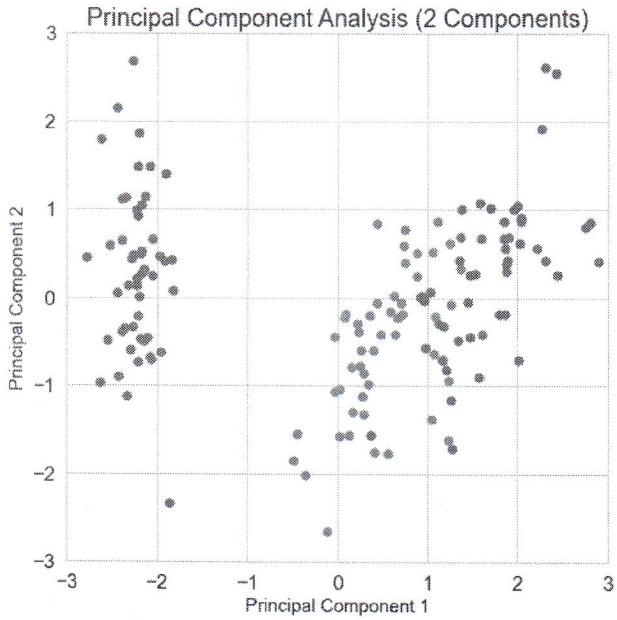
Principal component analysis is affected by attribute scale so we normalize all the attributes by eliminating the mean and scaling to unit variance.

In [65]:
```
x = StandardScaler().fit_transform(x)
```

In [66]:
```
pca = PCA(n_components=2)
new_data = pca.fit_transform(x)
pca_iris = pd.DataFrame(data = new_data,
                        columns = ['principal component 1', 'principal component 2'])
```

In [67]:
```
pca_iris_complete = pca_iris
pca_iris_complete['Species'] = iris[['Species']]
```

In [68]:
```
plt.figure(figsize = (8,8))
plt.xlabel('Principal Component 1', fontsize = 15)
plt.ylabel('Principal Component 2', fontsize = 15)
plt.title('Principal Component Analysis (2 Components)', fontsize = 20)
targets = dataset.target_names
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = pca_iris_complete['Species'] == target
    plt.scatter(pca_iris_complete.loc[indicesToKeep, 'principal component 1']
               , pca_iris_complete.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
# ax.legend(targets)
plt.axis([-3,3,-3,3])
plt.grid()
```



In [69]:
```
print("Explained Variance")
print("   Component 1 %3.2f"%(pca.explained_variance_ratio_[0]))
print("   Component 2 %3.2f"%(pca.explained_variance_ratio_[1]))
print("   Total Explained Variance %3.2f"%sum(pca.explained_variance_ratio_))
```

```
Explained Variance
   Component 1 0.73
   Component 2 0.23
   Total Explained Variance 0.96
```

with only 2 components we've explaining 96% of the variance

Rule of thumb: we stop considering variables when we explain 95% / 99% of the variance

In [70]:
```
print("Components")
for i,c in enumerate(pca.components_):
    print("Component %d\t%s"%(i,str(c)))
```

```
Components
Component 0     [ 0.52106591 -0.26934744  0.5804131   0.56485654]
Component 1     [0.37741762 0.92329566 0.02449161 0.06694199]
```

we can see that the first principal component considers all the variables with weights (0.52, −0.27, 0.58, 0.56) etc.

In [71]:
```
data = np.dot(x,np.transpose(pca.components_))
data[:5,:]
```

Out[71]:
```
array([[-2.26470281,  0.4800266 ],
       [-2.08096115, -0.67413356],
       [-2.36422905, -0.34190802],
       [-2.29938422, -0.59739451],
       [-2.38984217,  0.64683538]])
```

In [72]:
```
pca_iris.head(5)
```

Out[72]:

	principal component 1	principal component 2	Species
0	-2.264703	0.480027	setosa
1	-2.080961	-0.674134	setosa
2	-2.364229	-0.341908	setosa
3	-2.299384	-0.597395	setosa
4	-2.389842	0.646835	setosa

new dataset (projection on the new space)

We can apply PCA with the same number of components as the

In [73]:
```
x = iris.loc[:, numerical_variables].values
y = iris.loc[:,['Species']].values
full_pca = PCA()
fitted = full_pca.fit_transform(x)
full_pca.explained_variance_ratio_
```

Out[73]:
```
array([0.92461872, 0.05306648, 0.01710261, 0.00521218])
```

t-SNE

In [74]:
```
from sklearn.manifold import TSNE

perplexity=80
tsne = TSNE(n_components=2, verbose=1, perplexity=perplexity, n_iter=300, random_state=2867976)
# tsne = TSNE(n_components=2, verbose=1, perplexity=10, n_iter=300)
tsne_result = tsne.fit_transform(x)
```

```
[t-SNE] Computing 149 nearest neighbors...
[t-SNE] Indexed 150 samples in 0.000s...
[t-SNE] Computed neighbors for 150 samples in 0.003s...
[t-SNE] Computed conditional probabilities for sample 150 / 150
[t-SNE] Mean sigma: 1.546038
[t-SNE] KL divergence after 250 iterations with early exaggeration: 45.277550
[t-SNE] KL divergence after 300 iterations: 0.026498
```

In [75]:
```
iris_tsne = pd.DataFrame({'x':tsne_result[:,0], 'y':tsne_result[:,1], 'Species':iris['Species']})
```

In [76]:
```
iris_tsne
```

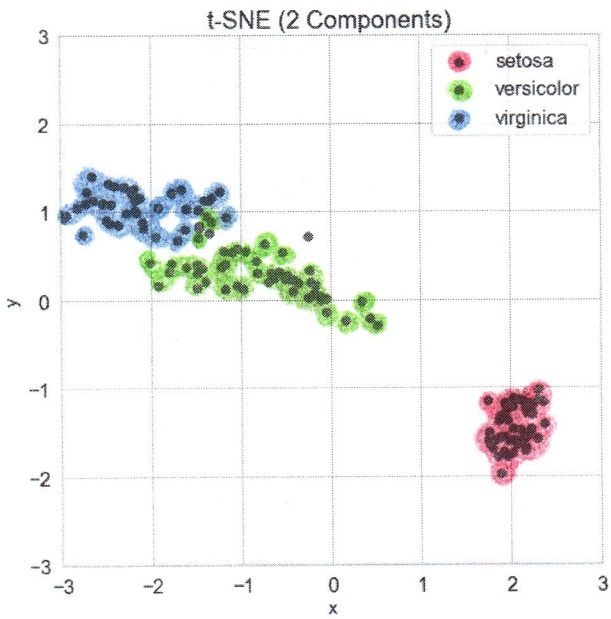
Out[76]:

	x	y	Species

	x	y	Species
0	2.058382	-1.646197	setosa
1	2.165277	-1.218571	setosa
2	2.199587	-1.170278	setosa
3	2.154776	-1.232397	setosa
4	2.119528	-1.608387	setosa
...	...	...	...
145	-2.262899	0.997264	virginica
146	-1.612844	1.038232	virginica
147	-2.096023	0.878660	virginica
148	-2.190123	1.250353	virginica
149	-1.477316	1.012344	virginica

150 rows × 3 columns

In [27]:
```
fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('x', fontsize = 15)
ax.set_ylabel('y', fontsize = 15)
ax.set_title('t-SNE (2 Components)', fontsize = 20)
targets = dataset.target_names
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = iris_tsne['Species'] == target
    ax.scatter(iris_tsne.loc[indicesToKeep,'x'], iris_tsne.loc[indicesToKeep,'y'], c=color, s=50)
ax.legend(targets)
plt.axis([-3,3,-3,3])
ax.grid()
```



House Prices: Advanced Regression Techniques (Part 1)

In this notebook we apply linear regression to some data from a Kaggle. The notebook is divided into two sections. First we perform some in-depth data exploration and pre-processing, next we build the actual models.

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

This notebook is derived from other existing notebooks by other authors, like,

- https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard

First we import some of the libraries we will need:

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import skew
from scipy.stats import norm
from scipy.stats.stats import pearsonr

%config InlineBackend.figure_format = 'retina' #set 'png' here when working on notebook
%matplotlib inline
```

In [2]:
```
train = pd.read_csv("HousePricesTrain.csv")
test = pd.read_csv("HousePricesTest.csv")
```

In [3]:
```
train.head()
```

Out[3]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	Mis
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	

5 rows × 81 columns

In [4]:
```
test.head()
```

Out[4]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	ScreenPorch	PoolArea	PoolQC	Fence
0	1461	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	AllPub	...	120	0	NaN	MnPrv
1	1462	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	NaN
2	1463	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	MnPrv
3	1464	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	AllPub	...	0	0	NaN	NaN
4	1465	120	RL	43.0	5005	Pave	NaN	IR1	HLS	AllPub	...	144	0	NaN	NaN

5 rows × 80 columns

→ Here we can add the command "train.describe(include="all")" / "test.describe(include="all")"

Note: There is No Class Label in the Test Set!

without the "include='all'" we would have only the numerical values ("test.describe()")

Note that the data set provided in a competition (and typically by a client) consist of a single batch of label data which we should use to build the model and possibly a set of "test" data without the target label. Your goal is to use the training set to develop a model to generate the target label for the data in the test set. You cannot use the test set for evaluating your model. In this case, we can only use the training dataset.

We generate a data set containing all the data except the class label for preprocessing purposes.

In [5]:
```
all_data = pd.concat((train.loc[:,'MSSubClass':'SaleCondition'],
                      test.loc[:,'MSSubClass':'SaleCondition']))
```

Data Exploration

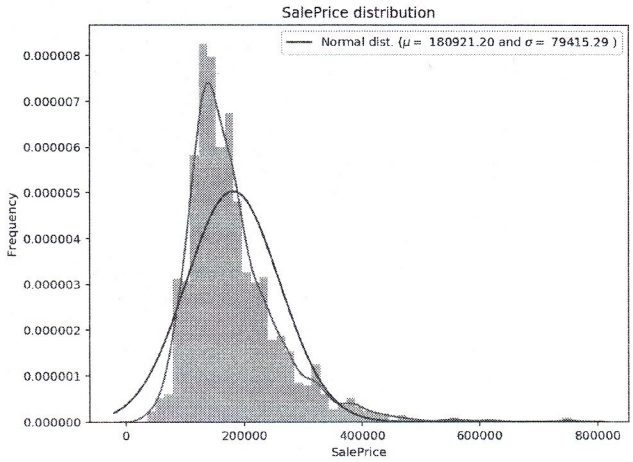
First, we should explore the data. We start from the target variable **SalePrice**, the variable we need to predict. We check its distribution.

In [6]:
```
matplotlib.rcParams['figure.figsize'] = (8.0, 6.0)
# fit the data with a normal distribution and
sns.distplot(train['SalePrice'] , fit=norm)

# and check the fitted mu and sigma
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
print("Skewness: %f" % train['SalePrice'].skew())
print("Kurtosis: %f" % train['SalePrice'].kurt())

plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
           loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')
```

```
 mu = 180921.20 and sigma = 79415.29
```

```
Skewness: 1.882876
Kurtosis: 6.536282
```

Out[6]: Text(0.5,1,'SalePrice distribution')



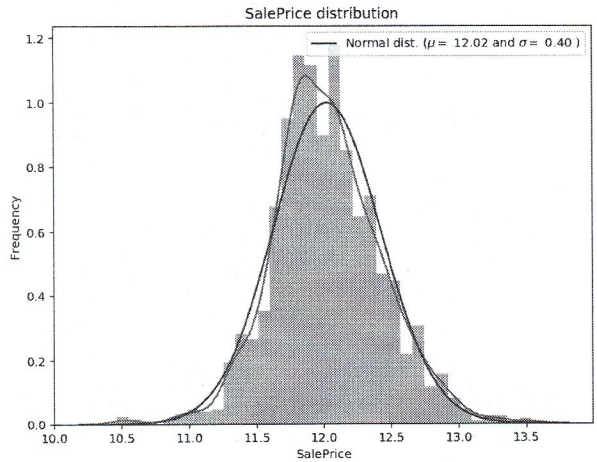
SalePrice has a skewed distribution. We can make **SalePrice** more normal by by taking log(**SalePrice** + 1) and this is generally a good practise.

In [7]:
```
#We use the numpy fuction log1p which  applies log(1+x) to all elements of the column
train["SalePrice"] = np.log1p(train["SalePrice"])
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
print("Skewness: %f" % train['SalePrice'].skew())
print("Kurtosis: %f" % train['SalePrice'].kurt())
```

```
 mu = 12.02 and sigma = 0.40

Skewness: 0.121347
Kurtosis: 0.809519
```

In [8]:
```
matplotlib.rcParams['figure.figsize'] = (8.0, 6.0)
sns.distplot(train['SalePrice'] , fit=norm)
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')
```

Out[8]: Text(0.5,1,'SalePrice distribution')



Missing Values

Let's check how many missing values are in the data set and how can we deal with them.

In [9]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

Here we've also sort the variables in such way that we look at the top-30 for missing values (the first) 30 variables that have more missing values)

Out[9]:

	Missing Ratio
PoolQC	99.657417
MiscFeature	96.402878
Alley	93.216855
Fence	80.438506
FireplaceQu	48.646797
LotFrontage	16.649538
GarageFinish	5.447071
GarageYrBlt	5.447071
GarageQual	5.447071
GarageCond	5.447071

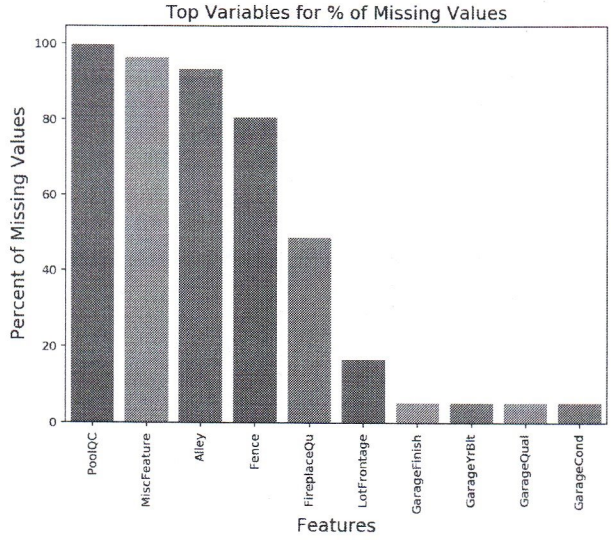
← 99,66% data missing (99.66% of "NULL")
Is this attribute completely useless?
FIRST OF ALL let's check for the meaning of "NaN" in this case. In the description of the data we can see that "PoolQC" = Pool quality and "NaN" corresponds to "No Pool" ⟹ MISSING VALUES ARE NOT REALLY MISSING!
Here it just mean that 99,66% of the people don't have a pool.

"MISSING VALUES" ARE NOT ALWAYS MISSING!
SOMETIMES ARE MISSING FOR A REASON!

In [10]:
```
f, ax = plt.subplots(figsize=(8,6))
```

```
plt.xticks(rotation='90')
sns.barplot(x=all_data_na.index[:10], y=all_data_na[:10])
plt.xlabel('Features', fontsize=15)
plt.ylabel('Percent of Missing Values', fontsize=15)
plt.title('Top Variables for % of Missing Values', fontsize=15)
```

Out[10]: Text(0.5,1,'Top Variables for % of Missing Values')



We note that some of the attributes have the vast majority of the values set to unknown. If we apply imputation without any knowledge about the meaning of what a missing value represent, we would end up with attributes set almost completely to the same value. So they would be almost useless.

How can we deal with all these missing values? First, we need to ask the domain expert whether some missing values have a special meaning. We actually don't have a domain expert but we have the data description in which we find out that

- Alley: Type of alley access to property, NA means "No alley access"
- BsmtCond: Evaluates the general condition of the basement, NA means "No Basement"
- BsmtExposure: Refers to walkout or garden level walls, NA means "No Basement"
- BsmtFinType1: Rating of basement finished area, NA means "No Basement"
- BsmtFinType2: Rating of basement finished area (if multiple types), NA means "No Basement"
- FireplaceQu: Fireplace quality, NA means "No Fireplace"
- Functional: data description says NA means typical
- GarageType: Garage location, NA means "No Garage"
- GarageFinish: Interior finish of the garage, NA means "No Garage"
- GarageQual: Garage quality, NA means "No Garage"
- GarageCond: Garage condition, NA means "No Garage"
- PoolQC: Pool quality, NA means "No Pool"
- Fence: Fence quality, NA means "No Fence"
- MiscFeature: Miscellaneous feature not covered in other categories, NA means "None"

Accordingly, we need to keep this information into account when dealing with the missing values of these variables.

Impute Categorical Values with Known Meaning

Let's impute the missing values for these attributes

In [11]:
```
all_data["Alley"] = all_data["Alley"].fillna("None")

# for all basement features a missing value means that there is no basement
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')

all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")

all_data["Functional"] = all_data["Functional"].fillna("Typ")

for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

all_data["PoolQC"] = all_data["PoolQC"].fillna("None")

all_data["Fence"] = all_data["Fence"].fillna("None")

all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
```

this means that in the variable "Alley" now we'll have 0 missing values (NAs), every "missing value" is replaced with "None". This is because we know that here the NAs have a meaning.

We can also deal with other numerical variables and use some heuristic to impute them. For instance, for **LotFrontage**, since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood , we can fill in missing values by the median LotFrontage of the neighborhood.

In [12]:
```
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
    lambda x: x.fillna(x.median()))
```

Take the data, group them by "Neighborhood" and consider the median of the "LotFrontage" for each Neighborhood. You are of the Neighborhood "A" and you have a missing value in "LotFrontage"? We assign you the value of the median of "LotFr." of the group "A".

We can also set the garage year, area and number of cars to zero for missing values since this means that there is no garage.

In [13]:
```
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
```

And we can do the same for basement measures.

In [14]:
```
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'BsmtHalfBath'):
```

```
    all_data[col] = all_data[col].fillna(0)
```

When **MasVnrArea** and **MasVnrType** are missing, it most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.

In [15]:
```
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

= MOST FREQUENT

For **MSZoning** (the general zoning classification), 'RL' is by far the most common value. So we can fill in missing values with 'RL'

In [16]:
```
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])
```

Utilities has all records are "AllPub", except for one "NoSeWa" and 2 missing values. Since the house with 'NoSewa' is in the training set, this feature won't help to predict labels in the test set. We can then safely remove it.

In [17]:
```
all_data = all_data.drop(['Utilities'], axis=1)
```

KitchenQual has only one missing value, and same as Electrical, we set it to the most frequent values (that is 'TA')

In [18]:
```
all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode()[0])
```

Exterior1st and **Exterior2nd** have only one missing value. We will use the mode (the most common value)

In [19]:
```
all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode()[0])
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode()[0])
```

For **SaleType** we can use the most frequent value (the mode) which correspond to "WD"

In [20]:
```
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])
```

For **MSSubClass** a missing value most likely means No building class. We can replace missing values with None

In [21]:
```
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

Electrical has one missing value. Since this feature has mostly 'SBrkr', we can set that for the missing value.

In [22]:
```
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0])
```

Anymore missing?

In [23]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

Out[23]:

	Missing Ratio

No more missing values! What would have happened if we did not use or did not have the data description?

Distribution of Numerical Variables

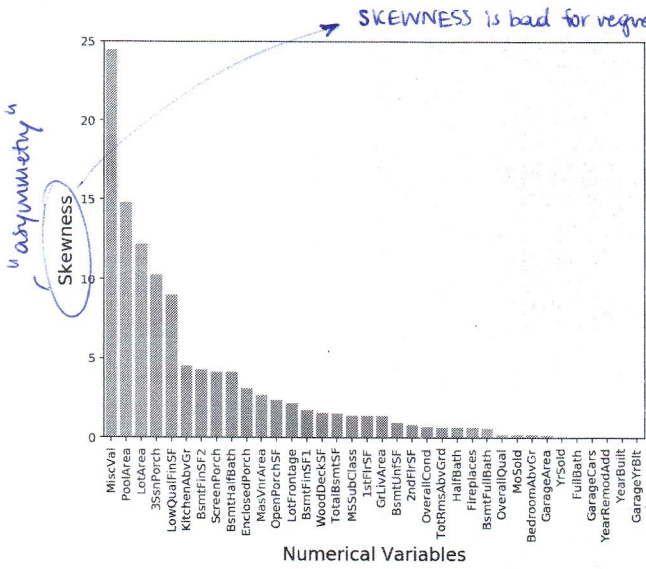
$\log 1p(data) = \log_e(data + 1)$

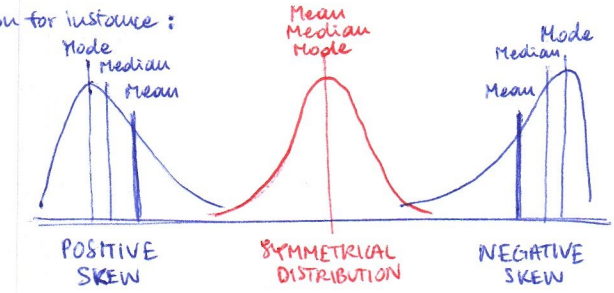
We now explore the distribution of numerical variables. As we did for the class, we will apply the log1p function to all the skewed numerical variables.

In [24]:
```
# take the numerical features
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index

# compute the skewness but only for non missing variables (we already imputed them but just in case ...)
skewed_feats = train[numeric_feats].apply(lambda x: skew(x.dropna()))

skewness = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.data})
# select the variables with a skewness above a certain threshold
```

In [25]:
```
skewness = skewness.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness['Variable'], y=skewness['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

Out[25]:
```
Text(0.5,1,'')
```



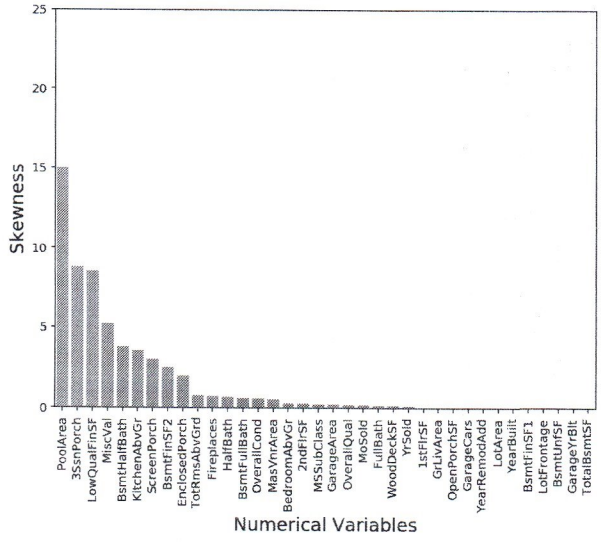


Let's apply the logarithmic transformation to all the variables with a skewness above a certain threshold (0.75). Then, replot the skewness of attributes. Note that to have a fair comparison the two plots should have the same scale.

In [26]:
```
skewed_feats = skewed_feats[skewed_feats > 0.75]
all_data[skewed_feats.index] = np.log1p(all_data[skewed_feats.index])
```

In [27]:
```
# compute the skewness but only for non missing variables (we already imputed them but just in case ...)
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))
skewness_new = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.data})
# select the variables with a skewness above a certain threshold

skewness_new = skewness_new.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness_new['Variable'], y=skewness_new['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

Out[27]:
```
Text(0.5,1,'')
```

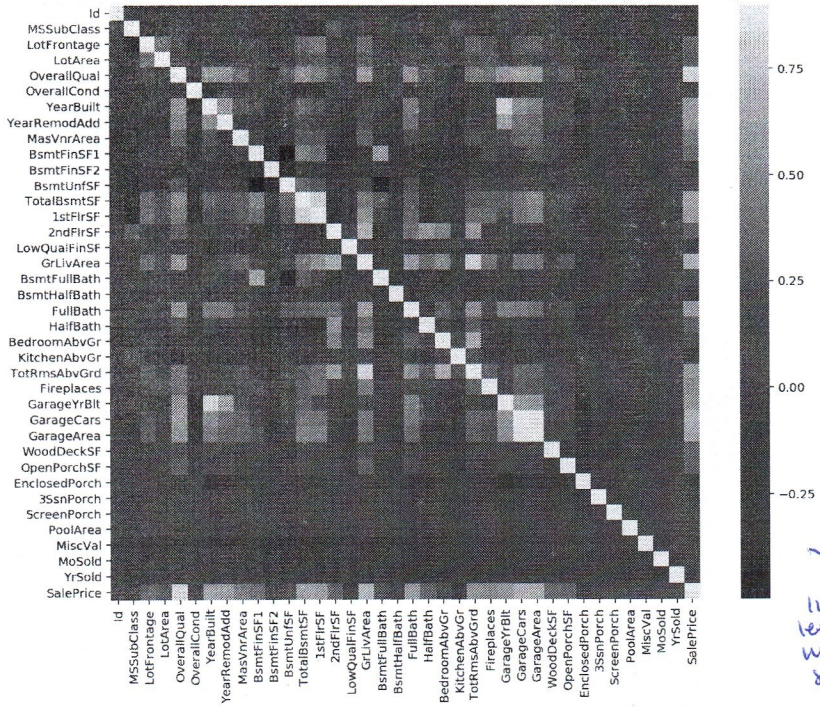


Correlation Analysis

We can finally perform some correlation analysis.

In [34]:
```
corrmat = train.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

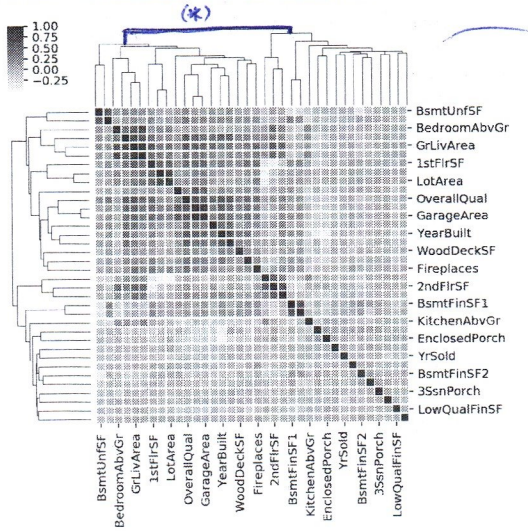
Out[34]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a243d4ef0>
```



Instead of looking for this let's do the clustering so that we'll see variables which has similar behaviour

In [29]:
```
import seaborn as sns
sns.clustermap(all_data.corr(), square=True, annot=False, cmap="Blues",
               linewidths=.75, figsize=(6, 6))
```

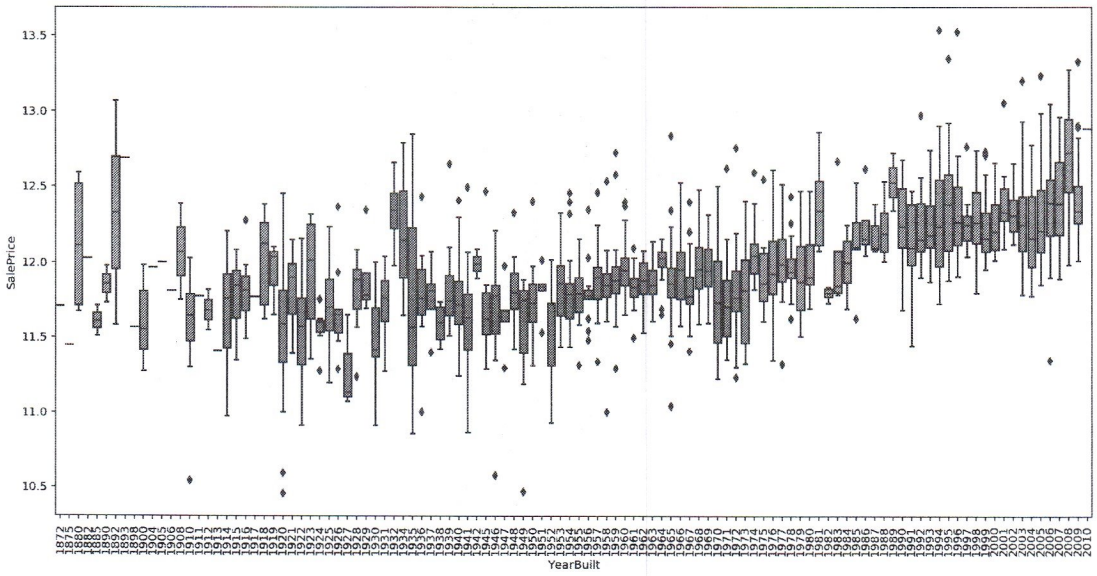
Out[29]: <seaborn.matrix.ClusterGrid at 0x1a2134f1d0>



From this clustering we see that there is a very "clear" grouping (2 groups of variable (*))

Or we can further analyze the distribution of some variables.

In [33]:
```
var = 'YearBuilt'
data = pd.concat([train['SalePrice'], train[var]], axis=1)
f, ax = plt.subplots(figsize=(16, 8))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
plt.xticks(rotation=90);
```



One Hot Encoding

We now generate the one hot encoding for all the categorical variables. Pandas has the function **get_dummies** that generates the binary variables for all the categorical variables

In [36]:
```
print("Number of Variables before OHE: "+str(all_data.shape[1]))
```

```
Number of Variables before OHE: 78
```

In [37]:
```
all_data = pd.get_dummies(all_data)
```

In [38]:
```
print("Number of Variables after OHE: "+str(all_data.shape[1]))
```

```
Number of Variables after OHE: 300
```

Saving the Preprocessed Data

We create the matrices to be used for computing the models and also save the cleaned data so that we can avoid repeating the process.

In [41]:
```
X_full = all_data[:train.shape[0]].copy()
X_full['SalePrice'] = train.SalePrice
X_full.to_csv("HousePricesTrainClean.csv")

# extract the test examples (we don't have the class value for this)
X_test = all_data[train.shape[0]:]

# save the test data to file
X_test.to_csv("HousePricesTestClean.csv")
```

London Bike Sharing

This is a preliminary analisys of the "London Bike Sharing" dataset. The goal is to practice exploratory data analysis with feature construction. The data comprise data from 1/1/2015 to 31/12/2016 and have been preprocessed from three original sources:

- Https://cycling.data.tfl.gov.uk/ 'Contains OS data © Crown copyright and database rights 2016' and Geomni UK Map data © and database rights [2019] 'Powered by TfL Open Data'
- freemeteo.com - weather data
- https://www.gov.uk/bank-holidays

The data from cycling dataset is grouped by "Start time", this represent the count of new bike shares grouped by hour. The long duration shares are not taken in the count.

There are ten attributes:

- timestamp: the timestamp
- cnt: number of bikes
- t1: actual temperature
- t2: temperature as it feels
- hum: humidity \%
- wind_speed: km/h
- weather_code: code
- is_holiday: boolean
- is_weekend: boolean
- season: season_code

The goal of the dataset is to predict the number of bikes that were rented at a certain hour of the day in London (Regression problem).

This is what we want to predict

In [1]:
```
import pandas as pd
import numpy as np
from datetime import date
import datetime
import matplotlib.pyplot as plt
%matplotlib inline

# Seaborn
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.set_context(rc={"font.family":'sans',"font.size":24,"axes.titlesize":24,"axes.labelsize":24})

# ignore the warnings
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
df = pd.read_csv("london_merged.csv")
df.head(5)
```

Since it's a code: it's actually NOMINAL

Out[2]:

	timestamp	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season
	NOM	N	N	N	N	N	N	B	B	N
0	2015-01-04 00:00:00	182	3.0	2.0	93.0	6.0	3.0	0.0	1.0	3.0
1	2015-01-04 01:00:00	138	3.0	2.5	93.0	5.0	1.0	0.0	1.0	3.0
2	2015-01-04 02:00:00	134	2.5	2.5	96.5	0.0	1.0	0.0	1.0	3.0
3	2015-01-04 03:00:00	72	2.0	2.0	100.0	0.0	1.0	0.0	1.0	3.0
4	2015-01-04 04:00:00	47	2.0	0.0	93.0	6.5	1.0	0.0	1.0	3.0

N = number (numerical var)
B = boolean (in this case the two booleans are NOMINAL)
NOM = nominal

First thing: what kind of data do we have?

Pay attention to those variables which are numbers but are NOMINAL!

Feature Creation

Timestamp encodes a lot of information that is unusable when contained in one variable. So first, let's extract all the information we might be interested in.

In [3]:
```
df['weekday'] = df['timestamp'].apply(lambda x: datetime.datetime.strptime(x,"%Y-%m-%d %H:%M:%S").weekday())
```

In [4]:
```
df['year'] = df['timestamp'].apply(lambda x: datetime.datetime.strptime(x,"%Y-%m-%d %H:%M:%S").year)
```

In [5]:
```
df['month'] = df['timestamp'].apply(lambda x: datetime.datetime.strptime(x,"%Y-%m-%d %H:%M:%S").month)
```

In [6]:
```
df['hour'] = df['timestamp'].apply(lambda x: datetime.datetime.strptime(x,"%Y-%m-%d %H:%M:%S").hour)
```

In [7]:
```
df['minute'] = df['timestamp'].apply(lambda x: datetime.datetime.strptime(x,"%Y-%m-%d %H:%M:%S").minute)
```

Let's check minutes, since I want to check whether the measurements are done hourly.

In [8]:
```
df['minute'].describe()
```

Out[8]:
```
count    17414.0
mean         0.0
std          0.0
min          0.0
25%          0.0
50%          0.0
75%          0.0
max          0.0
Name: minute, dtype: float64
```

Yes, they are hourly so let's eliminate this attribute and obviously also *timestamp*

In [9]:
```
df.drop(['minute','timestamp'],axis=1,inplace=True)
```

In [10]:
```
df.describe()
```

Notice that even if we have this, it does NOT mean that we can drop the "is_weekend" because being "5"/"6" (which correspond to the weekend) is not like being "weekend". The point is that in "weekday" the Saturdays and the Sundays are different, in the "weekend" not ⇒ different informations

! Note: how can we encode something periodic? Sundays are close to Mondays but 0s and 6s are not ⇒ sin(•) and cos(•) (or generally periodic) functions

However it's convenient to keep both the variable of the days and the periodi transf.

Out[10]:

	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season	weekday
count	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.00000
mean	1143.101642	12.468091	11.520836	72.324954	15.913063	2.722752	0.022051	0.285403	1.492075	2.99265
std	1085.108068	5.571818	6.615145	14.313186	7.894570	2.341163	0.146854	0.451619	1.118911	2.00406
min	0.000000	-1.500000	-6.000000	20.500000	0.000000	1.000000	0.000000	0.000000	0.000000	0.00000
25%	257.000000	8.000000	6.000000	63.000000	10.000000	1.000000	0.000000	0.000000	0.000000	1.00000
50%	844.000000	12.500000	12.500000	74.500000	15.000000	2.000000	0.000000	0.000000	1.000000	3.00000
75%	1671.750000	16.000000	16.000000	83.000000	20.500000	3.000000	0.000000	1.000000	2.000000	5.00000
max	7860.000000	34.000000	34.000000	100.000000	56.500000	26.000000	1.000000	1.000000	3.000000	6.00000

Also note that we have two temperatures, the real one t1 and the perceived one t2. It might be interesting to check the difference so that we can get extra information about the weather. If t2-t1 is higher, it means that the perceived temperature is higher, so it might depend on a warmer wind or else. Similarly if the difference is negative.

In [11]:
```
df['dt'] = df['t2'] - df['t1']
```

Other Ideas about how to explore and preprocess the data

- check for missing values => No missing values
- stats
- plot cnt with respect to the other attributes
- groupby weekday and plot ...
- % of cnt for each hour over day ...
- correlations ...

In [12]:
```
df.describe()
```

Out[12]:

	cnt	t1	t2	hum	wind_speed	weather_code	is_holiday	is_weekend	season	weekday
count	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.000000	17414.00000
mean	1143.101642	12.468091	11.520836	72.324954	15.913063	2.722752	0.022051	0.285403	1.492075	2.99265
std	1085.108068	5.571818	6.615145	14.313186	7.894570	2.341163	0.146854	0.451619	1.118911	2.00406
min	0.000000	-1.500000	-6.000000	20.500000	0.000000	1.000000	0.000000	0.000000	0.000000	0.00000
25%	257.000000	8.000000	6.000000	63.000000	10.000000	1.000000	0.000000	0.000000	0.000000	1.00000
50%	844.000000	12.500000	12.500000	74.500000	15.000000	2.000000	0.000000	0.000000	1.000000	3.00000
75%	1671.750000	16.000000	16.000000	83.000000	20.500000	3.000000	0.000000	1.000000	2.000000	5.00000
max	7860.000000	34.000000	34.000000	100.000000	56.500000	26.000000	1.000000	1.000000	3.000000	6.00000

We note that there are no missing values and cnt might be skewed

In [13]:
```
target_variable = 'cnt'
input_variables = df.columns[df.columns!=target_variable]
```

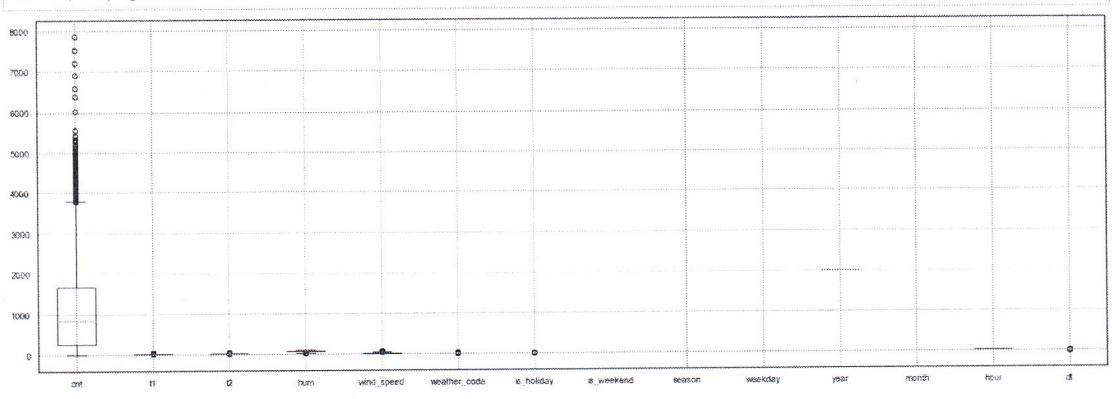
In [14]:
```
target_variable
```

Out[14]:
```
'cnt'
```

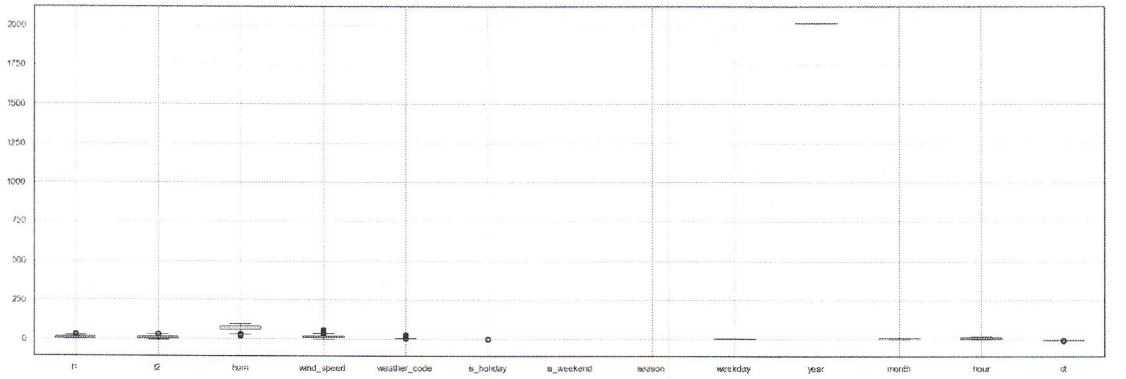
In [15]:
```
input_variables
```

Out[15]:
```
Index(['t1', 't2', 'hum', 'wind_speed', 'weather_code', 'is_holiday',
       'is_weekend', 'season', 'weekday', 'year', 'month', 'hour', 'dt'],
      dtype='object')
```

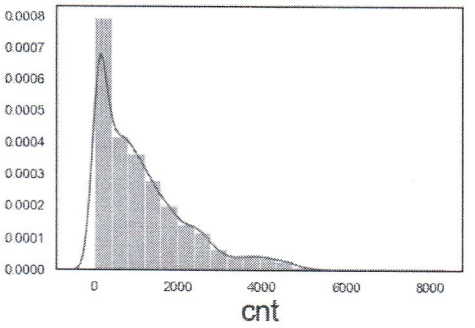
In [16]:
```
df.boxplot(figsize=(24,8));
```



In [17]:
```
df[input_variables].boxplot(figsize=(24,8));
```

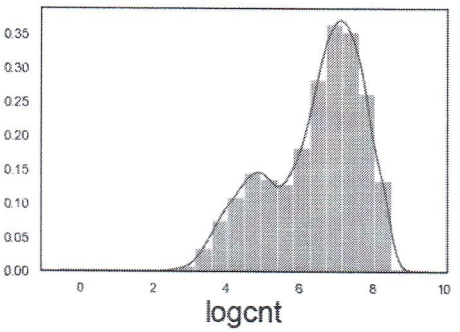


In [18]:
```
sns.distplot(df['cnt'], bins=20);
```



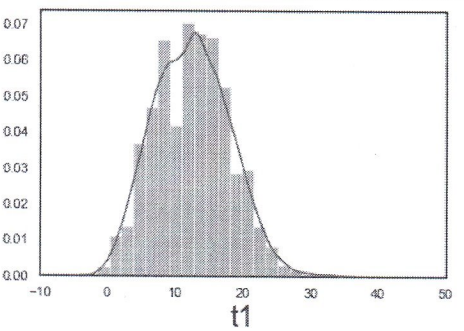
In [19]:
```
df['logcnt'] = np.log1p(df['cnt'])
```

In [20]:
```
sns.distplot(df['logcnt'], bins=20);
```

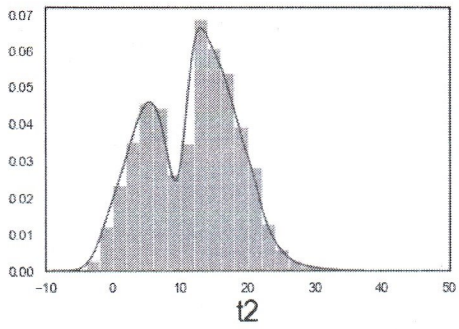


In [21]:
```
target_variable = 'logcnt'
```

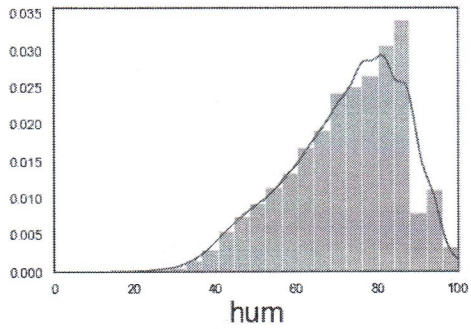
In [22]:
```
sns.distplot(df['t1'], bins=20);
plt.xlim([-10,50]);
```



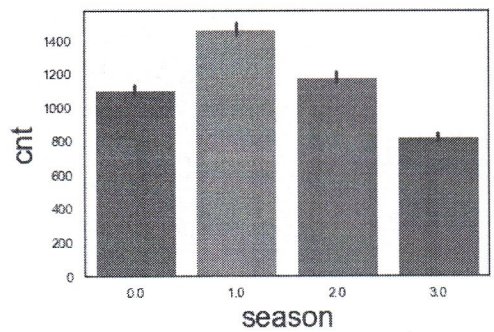
In [23]:
```
sns.distplot(df['t2'], bins=20);
plt.xlim([-10,50]);
```



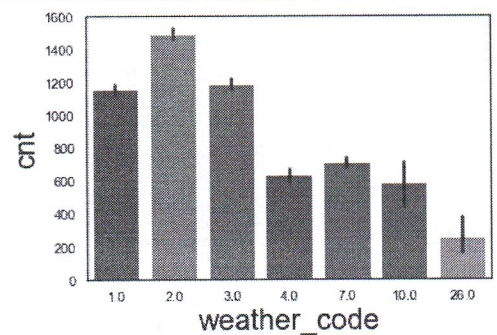
In [24]:
```
sns.distplot(df['hum'], bins=20);
plt.xlim([0,100]);
```



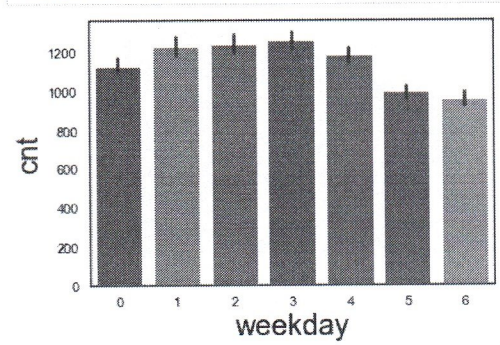
In [25]:
```
sns.barplot(x=df['season'],y=df['cnt']);
```



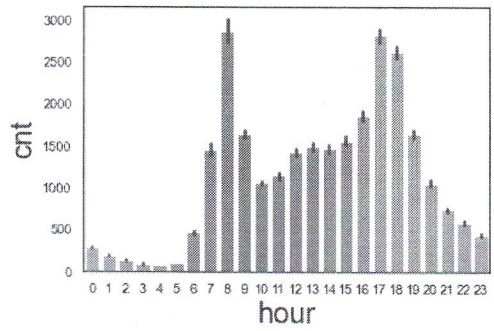
In [26]:
```
sns.barplot(x=df['weather_code'],y=df['cnt']);
```



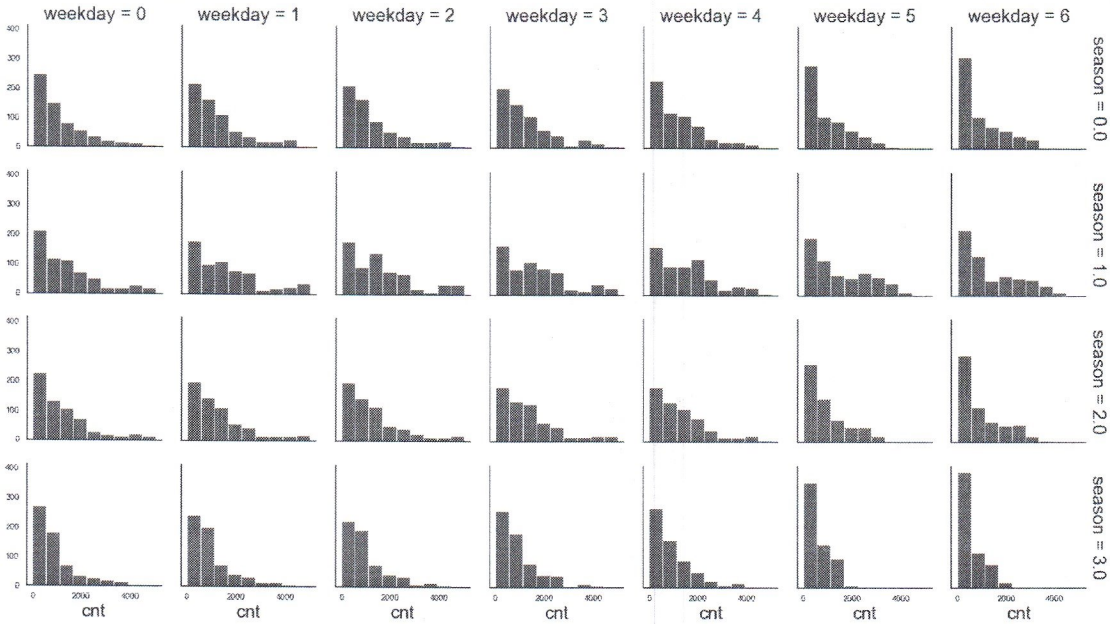
In [27]:
```
sns.barplot(x=df['weekday'],y=df['cnt']);
```



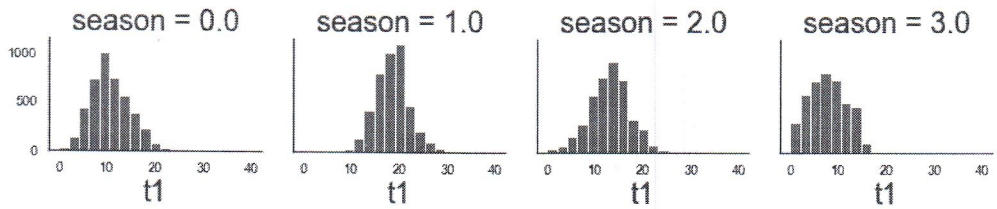
In [28]:
```
sns.barplot(x=df['hour'],y=df['cnt']);
```



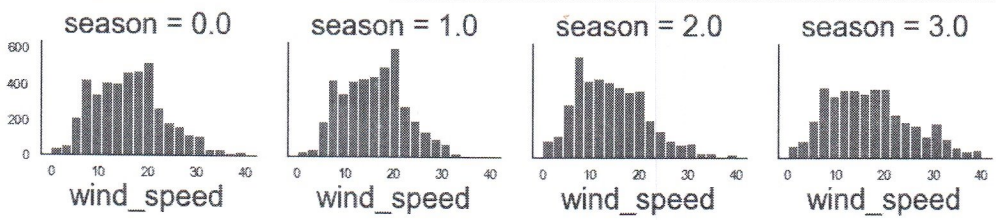
In [29]:
```
g = sns.FacetGrid(df, row='season', col='weekday',margin_titles=True);
g = g.map(plt.hist, "cnt", bins=np.linspace(0,5000,10));
```



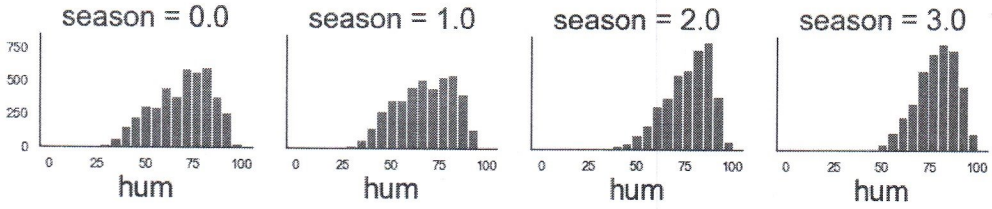
In [30]:
```
g = sns.FacetGrid(df, col='season', margin_titles=True);
g = g.map(plt.hist, "t1", bins=np.linspace(0,40,20));
```



In [31]:
```
g = sns.FacetGrid(df, col='season', margin_titles=True);
g = g.map(plt.hist, "wind_speed", bins=np.linspace(0,40,20));
```



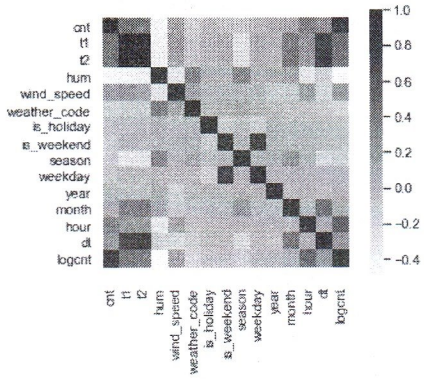
In [32]:
```
g = sns.FacetGrid(df, col='season', margin_titles=True);
g = g.map(plt.hist, "hum", bins=np.linspace(0,100,20));
```



In [33]:
```
#sns.pairplot(df);
```

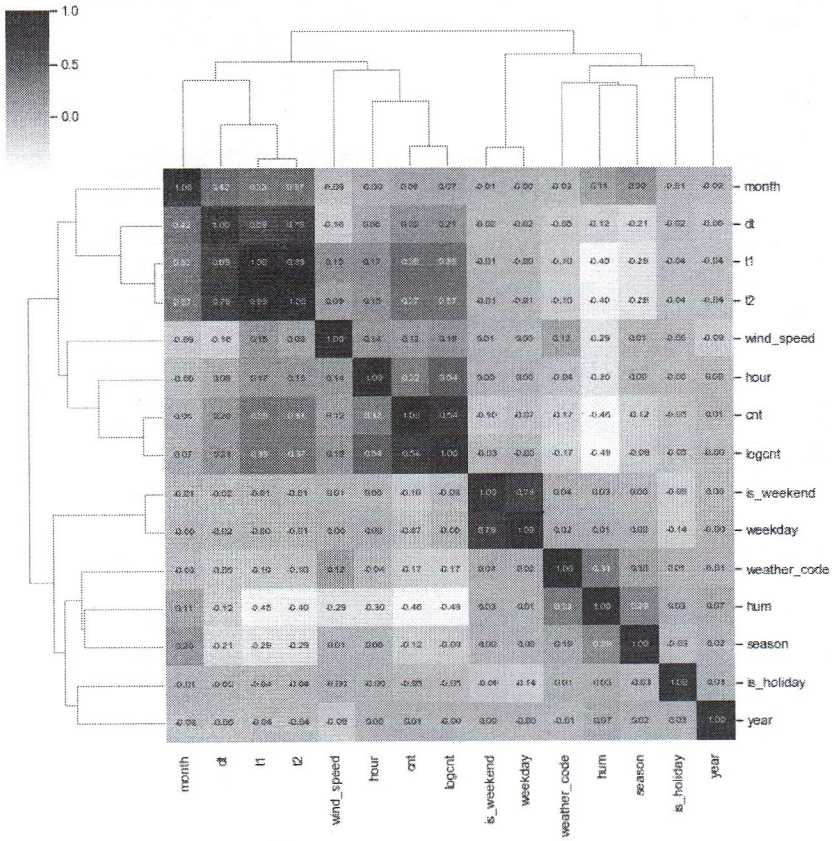
In [34]:
```
cov=df.corr(method='pearson')
sns.heatmap(cov,square=True,annot=False,cmap="Blues");
b, t = plt.ylim() # discover the values for bottom and top
b += 0.5 # Add 0.5 to the bottom
t -= 0.5 # Subtract 0.5 from the top
```

```
plt.ylim(b, t) # update the ylim(bottom, top) values
plt.show()
```

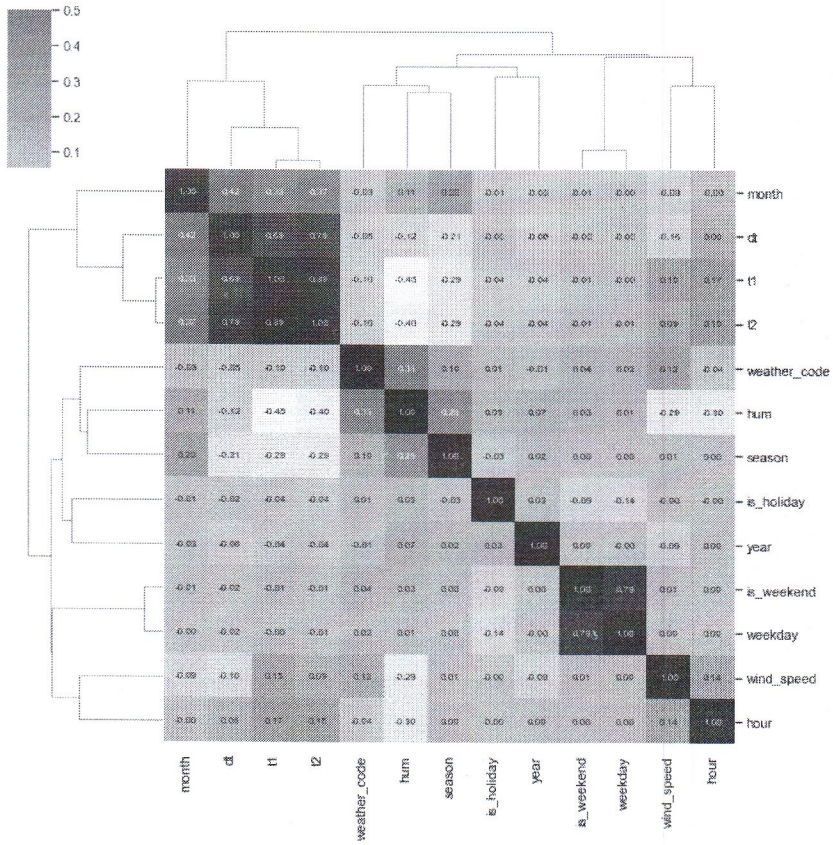


In [35]:
```
plt.figure(figsize=(15, 15));
sns.clustermap(cov,square=True,annot=True,cmap="Blues",annot_kws={"size": 8},fmt='.2f');
```

```
<Figure size 1080x1080 with 0 Axes>
```



In [36]:
```
cov2=df[input_variables].corr(method='pearson')
sns.clustermap(cov2,square=True,annot=False,cmap="Blues",annot_kws={"size": 8},fmt='.2f');
b, t = plt.ylim() # discover the values for bottom and top
b += 0.5 # Add 0.5 to the bottom
t -= 0.5 # Subtract 0.5 from the top
plt.ylim(b, t) # update the ylim(bottom, top) values
plt.show()
```



In [37]:
```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LassoCV
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
```

In [38]:
```
X = df[input_variables]
y = df[target_variable]
```

In [39]:
```
lr_model = LinearRegression();
lr_score = cross_val_score(lr_model,X,y,cv=KFold(n_splits=10, shuffle=True, random_state=1234))
```

In [40]:
```
lr_score.mean()
```

Out[40]: 0.4447209894684473

In [41]:
```
lr_score.std()
```

Out[41]: 0.02023832969066747

In [42]:
```
lasso_model = LassoCV();
lasso_score = cross_val_score(lasso_model,X,y,cv=KFold(n_splits=10, shuffle=True, random_state=1234))
```

In [43]:
```
lasso_score.mean()
```

Out[43]: 0.43776946564538016

In [44]:
```
lasso_score.std()
```

Out[44]: 0.020462238509046892

In [45]:
```
from sklearn.svm import SVR
svr_lin_model = SVR(kernel="linear")
svr_rbf_model = SVR(kernel="rbf")
```

In [46]:
```
#svr_lin_score = cross_val_score(svr_lin_model,X,y,cv=KFold(n_splits=10, shuffle=True, random_state=1234))
#svr_rbf_score = cross_val_score(svr_rbf_model,X,y,cv=KFold(n_splits=10, shuffle=True, random_state=1234))
```

In [47]:
```
from sklearn.neighbors import KNeighborsRegressor
knn_model = KNeighborsRegressor(n_neighbors=5, algorithm='kd_tree');
knn_score = cross_val_score(knn_model,X,y,cv=KFold(n_splits=10, shuffle=True, random_state=1234))
```

In [48]:
```
knn_score.mean()
```

Out[48]: 0.802673066886505

In [49]:
```
knn_score.std()
```

0.010229184332164987

Out[49]:

In [50]:
```
from sklearn.ensemble import RandomForestRegressor
rf_model = RandomForestRegressor(oob_score=True, random_state=1234);
```

In [51]:
```
rf_model.fit(X,y)
```

Out[51]:
```
RandomForestRegressor(oob_score=True, random_state=1234)
```

In [52]:
```
rf_model.oob_score_
```

Out[52]:
```
0.9685679568797737
```

In [ ]: