DSC - 01) Naive Bayes

Supervised classification technique that is based on Bayes Theorem with an assumption of independence among predictors

References:

> - Google Advanced Data Analytics Certificate
> - Gema Fernández, 2024 - Fundamentos de Ciencia de Datos en R (Book)

1.0)Theoretical background

1.0.1) Distances and MDS

Manhattan Distance

Also known as L1 distance or City Block distance, computes the absolute sum of the differences on each dimension betweeen the coordinates of two vectors. The name comes from the idea of a gridded city wher is only possible to transit the streets, and the distance between points A and B is calculated on such way.

For a p-dimensional space, the Mangattan Distance between elements i and j is computed as:

$$Manhattan\ Distance = \sum_{k=1}^{p} |X_{i,k} - X_{j,k}|$$

Multidimensional Scaling (MDS)

MDS is a technique used for dimension reduction and data visualization. When performing Metric MDS, the method tries to replicate the distances in a dimension-reduced space. When te algorithm returns negative eigenvalues, not all the information was able to be replicated. On the other hand, Non-Metric MDS uses a rank system, instead of distances or dissimilarities. It's used when working with categorical data.

Cosine Distance (not implemented)

The cosine distance it's a similiraty measurement between two vectors in a multidimensional space (two elements with multiple features). It's derived from the cosine of the angle between the vectors. The similiarty between the vectors i and j is computed as:

$$Cosine\ Similarity_{ij} = \frac{A \cdot B}{\|A\|\ \|B\|}$$

Therefore, the cosine distance between the vectors is obtained as:

$$D_{ij} = 1 - Cosine\ Similarity_{ij}$$

Note that the similiraty ranges between $(-1, 1)$ and, in consequence, the distance between $(0, 2)$. When the distance is -1, it means the vectors are completely opposited, when it's 0, the vectors are orthogonal, and when it's 1, the vectors point to the same direction.

1.0.2) Naive Bayes Classificator

Set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class clariable. The BT states the following:

$$P(y|\vec{x}) = \frac{P(y)P(\vec{x}|y)}{P(\vec{x})}$$

Assuming that $P(\vec{x})$ is constant and using the naive assumption of mutually independence among fetures, the expression is restated as the following expression. A Maximum A Posteriori (MAP) estimator $(\hat{y})$ is taken. Note that this it corresponds to the most frequented case in the dataset

$$P(y) \quad \alpha \quad P(y)\prod_{i=1}^{p} P(x_i|y)$$

$$\hat{y} = \underset{y}{max}\left\{P(y)\prod_{i=1}^{p} P(x_i|y)\right\}$$

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i|y)$. Note that this formula yields a probability, therefore, is needed to stablish a threshold to define wether an observation belongs to certain category or not. One of the biggest problems with Naive Bayes is the data assumptions. Few datasets have truly conditionally independence features. Despite of this shortback, and even when this assumption is violated, Naive Bayes models can still perform well. Another issue that could arise is what is known as the "zero-frequency" problem. This occurs when in the training dataset, is not found one of the categories of interest for a given factor. This would imply that the probability of observing such scenario is 0, ignoring the Cromwell's Principle. For such cases the workaround is adding an observation (+1 or +2) to each combination of predictor variables.

1.1) Prepare for execution

The datasets to be used is from the SciKit Learn library. It contains information about a sample of wines given their type.

In [557…

```python
# Basic libraries
import numpy as np
import pandas as pd
from scipy import stats

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style(style="ticks")
sns.set_palette("tab10")

# Used to perform multidimensional scaling
from sklearn.manifold import MDS
import scipy.spatial.distance as distance
from sklearn.metrics import pairwise

# Used for the ANOVA test
import statsmodels.api as sm
from statsmodels.formula.api import ols

# Used to implement Naive Bayes
import sklearn.preprocessing as preprocessing
from sklearn.model_selection import train_test_split
from sklearn import naive_bayes

# Load dataset
from sklearn import datasets
sk_wine = datasets.load_wine()
df_data = pd.DataFrame(data=sk_wine.data, columns=sk_wine["feature_names"])
df_data["wine_type"] = pd.Series(sk_wine["target"]).map({0:"cabernet", 1:"merlot", 2:"shiraz"})
df_data.rename({"od280/od315_of_diluted_wines": "protein"}, axis=1, inplace=True)
features = df_data.columns.to_list()
features.remove("wine_type")

# Block breaker
print("="*100, "\nTarget variable distribution in the dataset:","\n")
print(df_data["wine_type"].value_counts(normalize=True).round(3))
print("="*100)
```

```
====================================================================================================
Target variable distribution in the dataset:

wine_type
merlot      0.399
cabernet    0.331
shiraz      0.270
Name: proportion, dtype: float64
====================================================================================================
```

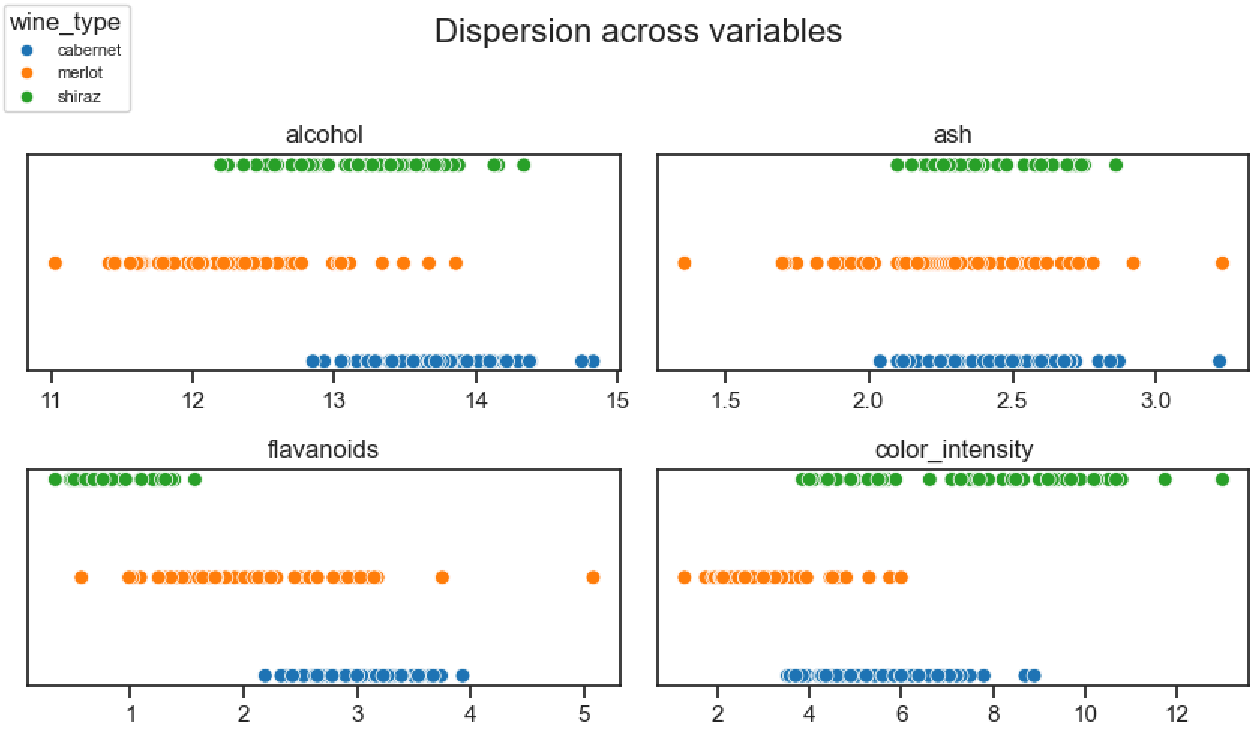
1.2) Exploratory data analysis

Three visualizations are provided:

1. Dispersion of variables given the wine type
2. Metric multidemnsional scaling
3. Correlation between explanatory variables

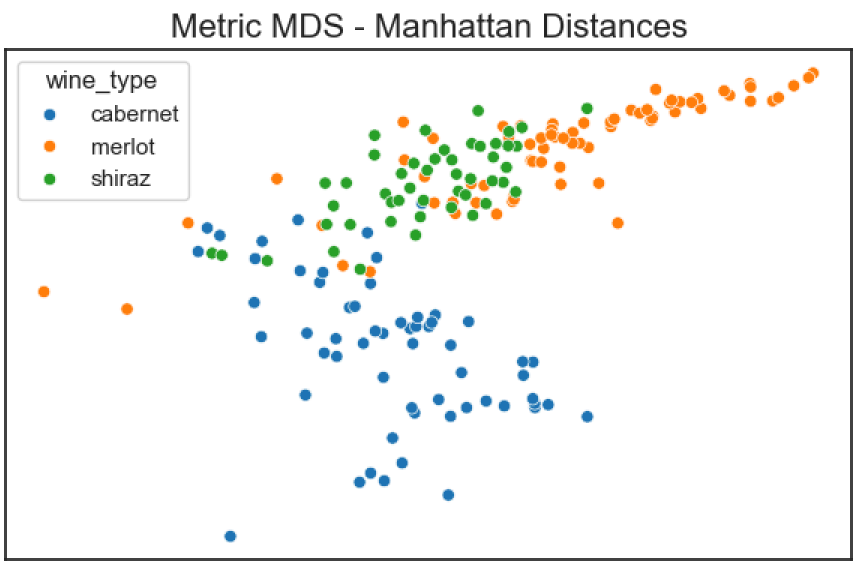
In [824…

```
# Used to extract legend to be displayed later
scatter_ax = plt.gca()
sns.scatterplot(data=df_data, x="alcohol", y="wine_type", hue="wine_type", ax=scatter_ax)
handles, labels = scatter_ax.get_legend_handles_labels()
plt.close()

# Plot grid of charts
fig, axes = plt.subplots(2, 2, figsize=(8.5,5))
variables = ["alcohol", "ash", "flavanoids", "color_intensity"]
for i, ax in enumerate(fig.axes):
    sns.scatterplot(df_data, x=variables[i], y="wine_type", ax=ax, hue="wine_type", legend=False, s=50)
    ax.set_title(variables[i], fontsize=12), ax.set_ylabel(""), ax.set_yticks([]), ax.set_xlabel(""), ax.s

# Add legend, adjust and show plot
plt.suptitle("Dispersion across variables", fontsize=16)
fig.legend(handles, labels, fontsize=8, loc="upper left", ncol=1, title='wine_type')
plt.tight_layout(rect=[0, 0, 1, .95])
plt.show()
```



In [445…

```
# Calculate NMDS based on city block distances
dissim = pairwise.manhattan_distances(df_data[features])
mds = MDS(n_components=2, dissimilarity="precomputed", metric=True, random_state=314159)
mds = mds.fit_transform(dissim)

# Plot results
plt.figure(figsize=(6,4))
sns.scatterplot(x=mds[:,0], y=mds[:,1], hue=df_data["wine_type"])
plt.title("Metric MDS - Manhattan Distances",fontsize=16), plt.xticks([]), plt.yticks([]), plt.tight_layou
plt.show()
```

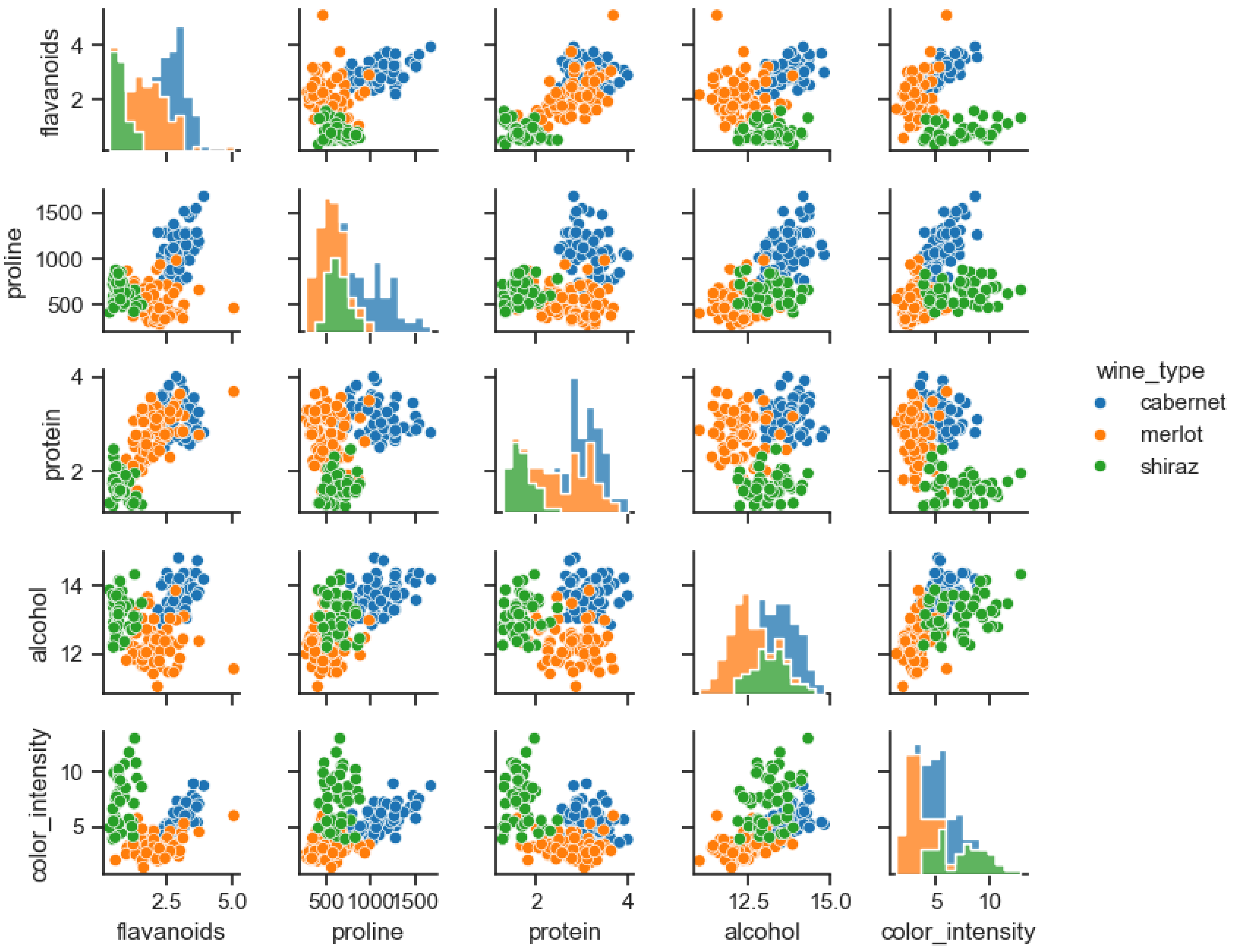


Before plotting the disperssion between variables, 5 of them are selected based on the p-value of an ANOVA analysis to test the significance of the means being the same. Prior to this process, the variables are standardized using a z-score method.

In [317…

```
# Scale features using a z-score
df_temporal = (df_data[features] - df_data[features].mean()) / df_data[features].std(ddof=1)
df_temporal = pd.concat([df_temporal, df_data["wine_type"]], axis=1)

# Adjust model and run one way ANOVA to select variables of interest
temporal = {}
for variable in features:
    model_ols = ols(formula=variable+" ~ C(wine_type)", data=df_data).fit()
    temporal[variable] = sm.stats.anova_lm(model_ols, typ=2).iloc[0,3]
variables = pd.Series(temporal).sort_values(ascending=True).index.to_list()[:5]

# Slice df to preserve only variables of interest
df_temporal = df_data[variables+["wine_type"]].copy()

# Correlation plot
grid = sns.PairGrid(df_temporal, hue="wine_type", height=1.5)
grid.map_diag(sns.histplot, multiple="stack", element="step", bins=15), grid.map_offdiag(sns.scatterplot)
grid.add_legend(), grid.fig.suptitle("Correlation of significant variables", fontsize=16), plt.tight_layou
plt.show()
```

Correlation of significant variables



1.3) Gaussian Naive Bayes (continuous variables)

For this case,the likelihood of the features is assumed to be Gaussian. Therefore, predictor variables can be continuous.

> $P(x_i|y) \sim N(\mu_i, \sigma_i)$

The parameters μ_i and σ_i are estimated using MLE.

1.3.1) Normalization of the predictors

In order to increase the normality of the variables, a normalization process is performed.

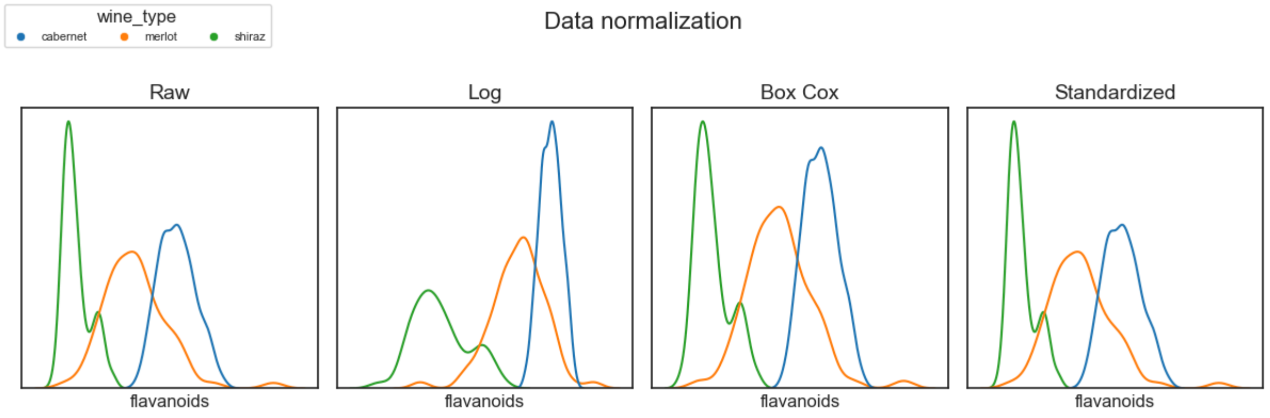
Notes:

> Observe that the improvement of normality is not as important as expected, due to the fact that raw data is normal enough.

In [657…

```
# Perform the data transformations
df_log = pd.concat([df_data[features].apply(np.log, axis=1), df_data["wine_type"]], axis=1)
df_boxcox = pd.concat([df_data[features].apply(lambda column: stats.boxcox(column)[0], axis=0), df_data["w
df_standardized = preprocessing.StandardScaler().fit_transform(df_data[features])
df_standardized = pd.DataFrame(data=standardized, index=df_data.index, columns=features)
df_standardized = pd.concat([df_standardized, df_data["wine_type"]], axis=1)

# Define variable of interest
variable = "flavanoids"
```

```
# Used to extract legend to be displayed later
scatter_ax = plt.gca()
sns.scatterplot(data=df_data, x="alcohol", y="wine_type", hue="wine_type", ax=scatter_ax)
handles, labels = scatter_ax.get_legend_handles_labels()
plt.close()

# Plot KDE of the variable for each transformation
fig, axes = plt.subplots(1,4, figsize=(12,4))
sns.kdeplot(x=variable, data=df_data,     hue="wine_type", ax=axes[0], legend=False, bw_adjust=0.75), axes[(
sns.kdeplot(x=variable, data=df_log,      hue="wine_type", ax=axes[1], legend=False, bw_adjust=0.75), axes[1
sns.kdeplot(x=variable, data=df_boxcox , hue="wine_type", ax=axes[2], legend=False, bw_adjust=0.75), axes[2
sns.kdeplot(x=variable, data=df_standardized, hue="wine_type", ax=axes[3], legend=False, bw_adjust=0.75), a

# Adjust settings for the subplots
axes[0].set_ylabel(""), axes[0].set_yticks([]), axes[0].set_xlabel(variable, fontsize=12), axes[0].set_xti
axes[1].set_ylabel(""), axes[1].set_yticks([]), axes[1].set_xlabel(variable, fontsize=12), axes[1].set_xti
axes[2].set_ylabel(""), axes[2].set_yticks([]), axes[2].set_xlabel(variable, fontsize=12), axes[2].set_xti
axes[3].set_ylabel(""), axes[3].set_yticks([]), axes[3].set_xlabel(variable, fontsize=12), axes[3].set_xti

# Display plots
# Add legend, adjust and show plot
plt.suptitle("Data normalization", fontsize=16)
fig.legend(handles, labels, fontsize=8, loc="upper left", ncol=3, title='wine_type')
plt.tight_layout(rect=[0, 0, 1, .95])
plt.show()
```



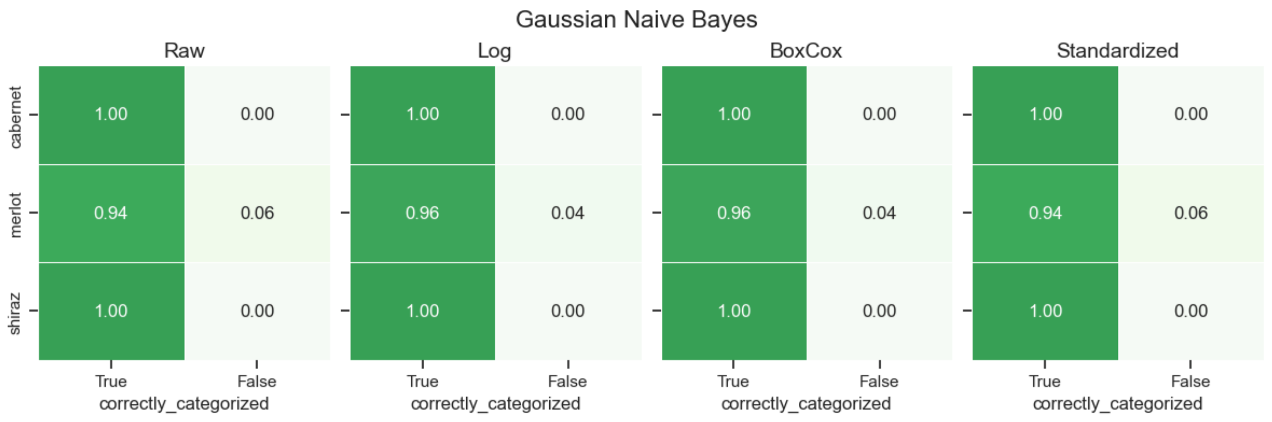
1.3.2) Implementation for classification purposes

The response variable can be any of the wine types in the dataset.

In [793…

```
# Fit model for each data transformation
fig, axes = plt.subplots(1, 4, figsize=(12,4), sharey=True)
datasets = [df_data[features], df_log[features], df_boxcox[features], df_standardized[features]]
transformations = ["Raw", "Log", "BoxCox", "Standardized"]
for i, ax in enumerate(fig.axes):

    # Split dataset in train and test subsets
    X = datasets[i].values
    y = df_data["wine_type"]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.70, stratify=y, random_state=314

    # Fit model and predict for test data
    model_gnb = naive_bayes.GaussianNB().fit(X_train, y_train)
    y_pred = model_gnb.predict(X_test)

    # Plot confussion matrix
    cmx = pd.DataFrame(np.nan, index=df_data["wine_type"].unique(), columns=["True", "False"])
    cmx.index.name = "wine_type"
    cmx.columns.name = "correctly_categorized"
    for type in df_data["wine_type"].unique():
        cmx.loc[type, "True"] = ((y_test==type) * (y_pred==type)).sum()
        cmx.loc[type, "False"] = (y_test==type).sum() - cmx.loc[type, "True"]
    cmx = (cmx.T / y_test.value_counts()).T
    sns.heatmap(cmx, vmin=0, vmax=1.5, cmap="Greens", ax=ax, annot=True, fmt="0.2f", cbar=False, linewidth
    ax.set_title(transformations[i], fontsize=14), ax.set_ylabel("")
```

```python
# Display figure
plt.tight_layout(rect=(0,0,1,0.95)), plt.suptitle("Gaussian Naive Bayes", fontsize=16)
plt.show()
```



1.4) Multinomial Naive Bayes (categorical variables)

Assumes data follows a multinomial distribution. The distribution is parametrized by vectors θ_y for each class y, where n is the number of featuresand $\theta_{(y,i)}$ is the probability $P(x_i|y)$ (probability of feature i appearing in a sample belonging to class y)

> $$\theta_y = (\theta_{(y,1)}, \theta_{(y,2)}, \ldots, \theta_{(y,n)})$$

The parameters $\theta_{(y,i)}$ are estimated by a smoothed bersion of ML relative to the frequency counting. Where $N_{(y_i)} isthenumberoftimesfeature$i\$\$ appears ina sample of class y in the training set and $N_{(y,i)}$ is the total count of all features for class y. THe smoothing priors $\text{\alopha} \geq 0$ accounts fo features not present in the learning smamples and prevents zero probabilities in further computations. Settin $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.

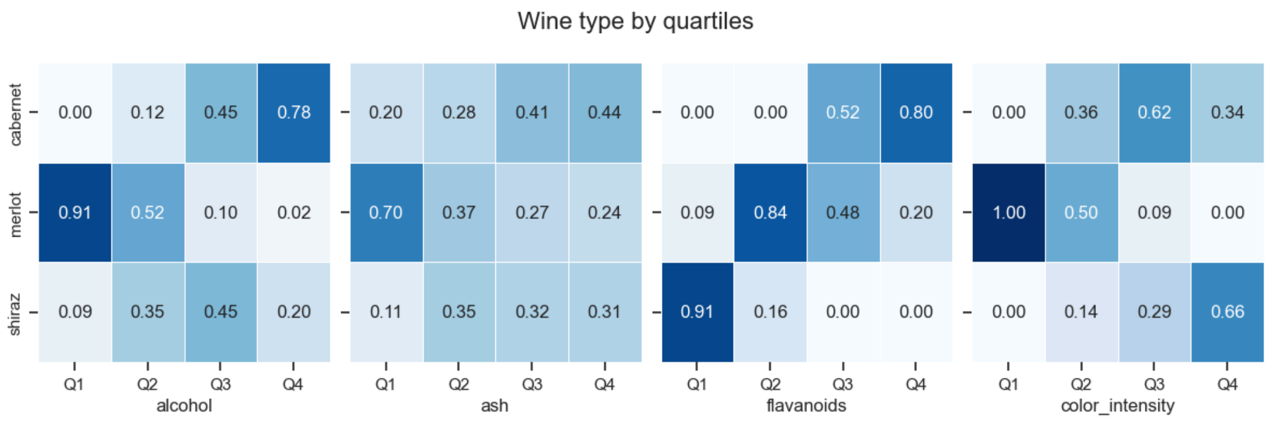
> $$\hat{\theta}_{(y,i)} = \frac{N_{(y,i)} + \alpha}{N_y + \alpha n}$$

1.4.1) Data transformations

Note that all the variables are continuous. In order to trahnsform them into categorical variables, a percentil-based approach is used, dividing the range in 4 levels.

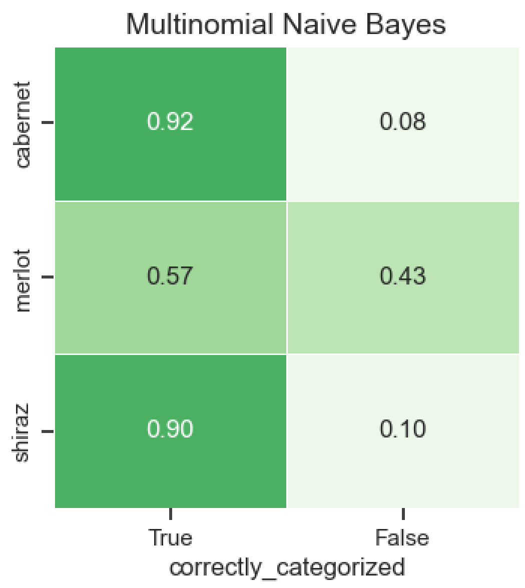
In [739…

```python
# Define a function to split in quartiles a numerical vector
def categorize(values):
    values = np.array(values)
    results = np.full(values.shape[0], np.nan)
    results = np.where(values <= np.percentile(values, 25), "Q1", results)
    results = np.where((values > np.percentile(values, 25)) & (values <= np.percentile(values, 50)), "Q2",
    results = np.where((values > np.percentile(values, 50)) & (values <= np.percentile(values, 75)), "Q3",
    results = np.where((values > np.percentile(values, 75)) & (values <= np.percentile(values, 100)), "Q4"
    return results

# Create a df with quartiles instead of values
df_quartiles = df_data.copy()
df_quartiles[features] = df_data[features].apply(categorize)

# Create plot grid
fig, axes = plt.subplots(1,4,figsize=(12,4), sharey=True)

# Plot a heatmap of wine type given the quartiles in variables of interest
for i, ax in enumerate(fig.axes):
    variable = variables[i]
    contingency_table = pd.crosstab(df_quartiles["wine_type"], df_quartiles[variable])
    contingency_table = (contingency_table / contingency_table.sum(axis=0))
    sns.heatmap(contingency_table, annot=True, fmt='.2f', cmap="Blues",cbar=False, vmin=0, vmax=1 ,ax=ax,

# Show plot
plt.suptitle("Wine type by quartiles", fontsize=16)
plt.tight_layout()
plt.show()
```



1.4.2) Implementation for classification purposes

```
In [808…
# Split dataset in train and test subsets
X = pd.get_dummies(df_data[features]).values
y = df_data["wine_type"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=314159)

# Fit model and predict for test data
model_gnb = naive_bayes.MultinomialNB(alpha=2).fit(X_train, y_train)
y_pred = model_gnb.predict(X_test)

# Plot confussion matrix
cmx = pd.DataFrame(np.nan, index=df_data["wine_type"].unique(), columns=["True", "False"])
cmx.index.name = "wine_type"
cmx.columns.name = "correctly_categorized"
for type in df_data["wine_type"].unique():
    cmx.loc[type, "True"] = ((y_test==type) * (y_pred==type)).sum()
    cmx.loc[type, "False"] = (y_test==type).sum() - cmx.loc[type, "True"]
cmx = (cmx.T / y_test.value_counts()).T
fig, ax = plt.subplots(1,1, figsize=(4,4))
sns.heatmap(cmx, vmin=0, vmax=1.5, cmap="Greens", ax=ax, annot=True, fmt="0.2f", cbar=False, linewidth=.5)
ax.set_title("Multinomial Naive Bayes", fontsize=14), ax.set_ylabel("")
plt.show()
```


Multinomial Naive Bayes
cabernet
merlot
shiraz
True
False
correctly_categorized