2.2.3 Naive Bayes

The third and final classification algorithm we explore is the **Naive Bayes** algorithm. As we mentioned previously, classification problems can be solved by creating a predictive model. That is what we have done with *Logistic Regression*. Another way to create a predictive model would be to **estimate** the conditional probability of the class label, given the observation.This means we can calculate the conditional probability for each class label and then pick the label with the highest probability as the most likely label. In theory, Bayes Theorem can be used for this:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

The main problem with this approach is that we need a really large dataset to calculate the **conditional probability** *P(x1, x2, …, xn | yi)*, because this formula assumes that each input variable is dependent upon all other variables. If the number of features is large, the size of the dataset becomes an even bigger problem. To simplify this problem, we **assume** that each input variable is independent of the other. This might sound weird…because it is. In reality, it is really rare that the input features don't **depend** on each other. However, this approach proved to do surprisingly well in the wild. That is why we can rewrite the formula from above as:

$$P(y_i|x_1, x_2, \ldots, x_n) = \frac{P(x_1, x_2, \ldots, x_n|y_i) * P(y_i)}{P(x_1, x_2, \ldots, x_n)}$$

To calculate *P(yi)* all we have to do is divide the frequency of class *yi* in the training dataset and divide it with the total number of samples in the training set *(P(yi) = # of samples with yi / total # of examples)*. The second part of the **equation**, the conditional probability, can be derived from the data as well. So, let's implement it.

2.2.3.1 Preparing Data for Naive Bayes

Before we dive into the algorithm implementation, let's prepare the *PalmerPenguins* dataset again. It is very similar for the preparation we have done for KNN:

```
data = pd.read_csv('./data/penguins_size.csv')
ss = StandardScaler()

data = data.dropna()
data = data.drop(['sex', 'island', 'flipper_length_mm', 'body_mass_g'], axis=1)

# Prepare input
X = data.drop(['species'], axis=1)
columns = X.columns
X = X.values
X = ss.fit_transform(X)

# Prepare target
y = data['species']
species = {'Adelie': 0, 'Chinstrap': 1, 'Gentoo': 2}
y = [species[item] for item in y]
y = np.array(y)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=33)
```

The only difference is that we stored columns in the *columns* variable because we need them for the algorithm implementation.

2.2.3.2 Python Implementation

The implementation of this algorithm is simple and can be found in the class *MyNaiveBias*. Note that we utilized the *Pandas* library a bit more because it provides easy solutions for **grouping** data:

```
class MyNaiveBias():
    '''Implements algorithm for Naive Bias'''
    def __init__(self, input_columns):
        self.input_columns = input_columns

    def fit(self, X_train, y_train):
        X_train = pd.DataFrame(X_train, columns = self.input_columns)
```

```
        self.classes = np.unique(y_train)
        self.means = X_train.groupby(y_train).apply(np.mean)
        self.stds = X_train.groupby(y_train).apply(np.std)
        self.probabilities = X_train.groupby(y_train).apply(lambda x: len(x)) /
X_train.shape[0]

    def predict(self, X_test):
        X_test = pd.DataFrame(X_test, columns = self.input_columns)
        predictions = []

        for i in range(X_test.shape[0]):
            p = {}

            for c in self.classes:
                p[c] = self.probabilities[c]

                for index, row in enumerate(X_test.iloc[i]):
                    p[c] *= norm.pdf(row, self.means.iloc[c, index],
self.stds.iloc[c, index])

            predictions.append(pd.Series(p).values.argmax())

        return predictions
```

So, the fit method is calculating **statistics** for the training dataset. We store all classes, calculate means and standard deviations for the input training data grouped by each class. Finally, we calculate the prior probability for each class. The *predict method* is doing all the hard work here and utilizes those calculations. First, for each sample from the input set and for each class, we calculate the **conditional probability** and multiply it by the **prior probability** for that class. In the end, for each input sample, we pick the class with the **largest** probability. We use this class as the ones before:

```
model = MyNaiveBias(columns)

model.fit(X_train, y_train)
my_naive_predictions= model.predict(X_test)

print(metrics.classification_report(y_test, my_naive_predictions))

pd.DataFrame({
    'Actual Value': y_test,
    'Naive Bias Predictions': my_naive_predictions,
})
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        25
           1       0.83      1.00      0.91        10
           2       1.00      0.94      0.97        34

    accuracy                           0.97        69
   macro avg       0.94      0.98      0.96        69
weighted avg       0.98      0.97      0.97        69
```

	Actual Value	Naive Bias Predictions
0	1	1
1	2	2
2	0	0
3	2	2
4	0	0
...	...	...
62	2	2
63	2	2
64	2	2
65	0	0
66	2	2

2.2.3.3 Using SciKit Learn

We can use *SciKit Learn* implementation of Naive Bayes like this:

```
model = GaussianNB()
model.fit(X_train, y_train)
sk_nb_preditions = model.predict(X_test)

print(accuracy_score(sk_nb_preditions, y_test))

pd.DataFrame({
    'Actual Value': y_test,
    'Naive Bias Predictions': predictions,
    'SciKit Learn NB': sk_nb_preditions,
})
```