Dog or Not

A comprehensive classifier analysis on the association of vendor names with its description

by Zaki Aslam, Hector Palafox Prieto, Jennifer Tsang, and Samrawit Mezgebo Tsegay

In [1]:

```python
import numpy as np
import pandas as pd
import altair as alt
import matplotlib.pyplot as plt
import pandera.pandas as pa
from sklearn.model_selection import (
    train_test_split, cross_validate,
    cross_val_predict, RandomizedSearchCV
)
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.impute import SimpleImputer
from sklearn.compose import make_column_transformer
from sklearn.dummy import DummyClassifier
from sklearn.naive_bayes import BernoulliNB
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.linear_model import LogisticRegression
from scipy.stats import loguniform, randint
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from mglearn.tools import visualize_coefficients
```

Summary

In this project, we used decision trees, logistic regression, and a Naive Bayes classifier to identify whether or not a food vendor sells hot dogs via their name. We trained each model individually using a cross-validation setup, and we compared the scores of the accuracy in order to determine a model to train and to compare to the test data. The model we chose, finally, was the Naive Bayes, as it provided a slightly better score and less underfit and overfit than the other models present. Finally, we validated it with our test data and came to the conclusion that even though it is good enough for classifying most of the cases, it still struggles discerning from the minority class, which in our case, is our target.

Introduction

Food trucks and mobile food vendors are a common sight in Downtown Vancouver, offering a wide range of cuisine types from hot dogs and burgers to shawarma and tacos. With so many different vendors and food options, it can be useful to automatically identify what kind of food a vendor specializes in based only on select information. In this project, we study whether we can predict if a food vendor is a hot dog vendor or not using the vendor's business name. We used a publicly available dataset of mobile food vendors in Vancouver from the City of Vancouver's open data portal, where each row represents relevant information for a single food vendor and includes columns such as BUSINESS_NAME, LOCATION, DESCRIPTION, GEO_LOCALAREA, and geographic coordinates. For our analysis, we constructed a binary target variable named is_hotdog, which is True when the DESCRIPTION is "Hot Dogs" and False otherwise. This allows us to investigate how much information about the type of food a vendor sells can be extracted from the business name, as well as putting to test the prediction power of some of the most common classification algorithms: Decision Trees, Logistic Regression, and Naïve-Bayes.

Methods & Results

Data

We are using the data directly from the Vancouver City hall portal, yet an offline copy is present on the `data` directory, retrieved on **2025-11-17**.

- **KEY** (`str`): A unique identifier of the vendor
- **VENDOR_TYPE** (`str`): The type of vendor for that specific location (in our case all of them are vendor_food)
- **STATUS** (`str`): Whether or not the vendor remains open or not (in our case all of them are open)
- **BUSINESS_NAME** (`str`): The name of the vendor
- **LOCATION** (`str`): Address of the vendor
- **DESCRIPTION** (`str`): The category of food offered by the vendor
- **GEO_LOCALAREA** (`str`): The zone/area of the city in which the vendor is located
- **Geom** (`dict`): The coordinates stored as a JSON object
- **geo_point_2d** (`2d_array`): The coordinates stored as a 2D array

Data validation

Before splitting the data into training and test sets and fitting models, we perform basic data validation on the raw tabular data to check that it is well-formed and consistent with our expectations.

In particular, we check:

- that the data file can be read and has the expected tabular format
- that the expected columns are present and correctly named
- that there are no completely empty rows
- that missing values are within an acceptable range
- that each column has an appropriate data type for our analysis

Check 1: Data file format

We expect the food-vendors data to come from a CSV file that can be read into a non-empty pandas DataFrame. If the download fails or the file is not in the expected tabular format, we want the analysis to stop early instead of producing confusing errors later.

In [2]:

```python
## Check 1: data file format (tabular CSV)

## loads data from the original source on the web
url = (
    "https://opendata.vancouver.ca/api/explore/v2.1/"
    "catalog/datasets/food-vendors/exports/"
    "csv?lang=en&timezone=America%2FLos_Angeles"
    "&use_labels=true&delimiter=%3B"
)

try:

    ## We sort the data by KEY in order to be consistent
    ## if source order changes.
    food_vendors = pd.read_csv(url, sep = ";").sort_values("KEY")
    food_vendors.head(5)
```

```
except:
    raise FileNotFoundError(
        "Data loading failed: The link to the data source is likely broken"
    )
```

In [3]:

```
food_vendors.head(5)
```

Out[3]:

	KEY	VENDOR_TYPE	STATUS	BUSINESS_NAME	LOCATION	DESCRIPTION	GEO_LOCALAREA	
10	C1	vendor_food	open	Vie Niapolotan	Authorised Parking Meter - West Side of 400 Bu...	Italian Cuisine	Downtown	{"coordi [-123.1175335 49.286
73	C10	vendor_food	open	Mum's Grilled Cheese Truck	Authorised Parking Meter - West Side of 700 How...	Grilled Cheese and Soups	Downtown	{"coordi [-123.1206849 49.282
11	C11	vendor_food	open	Meet to Eat	Authorised Parking Meter South Side of 1000 Du...	Kosher Israeli	Downtown	{"coordi [-123. 49.285658],
1	C12	vendor_food	open	Chickpea	Authorised Parking Meter - North Side of W Cor...	Vegetarian	Downtown	{"coordi [-123.1157734 49.287
68	C13	vendor_food	open	Disco Cheetah	Authorised Parking Meter West Side of 700 Howe St	Korean	Downtown	{"coordi [-123.120443 49.2823

Table 1: Raw data from web source

In [4]:

```
assert isinstance(food_vendors, pd.DataFrame), (
    "Expected `food_vendors` to be a pandas DataFrame."
)

n_rows, n_cols = food_vendors.shape
assert n_rows > 0 and n_cols > 0, (
    "Downloaded data appears to be empty (no rows or no columns)."
)

print(f"Data format check passed: {n_rows} rows × {n_cols} columns.")
```

```
Data format check passed: 91 rows × 9 columns.
```

Check 2: column names

For our analysis we only rely on the `BUSINESS_NAME` and `DESCRIPTION` columns. Here we check that these columns are present in the downloaded table and have the expected names. If either column is missing or renamed, we stop the analysis and fix the data import first.

In [5]:

```
## Check 2: column names (only columns used in the analysis)

required_columns = ["BUSINESS_NAME", "DESCRIPTION"]
actual_columns = food_vendors.columns.tolist()

missing_required = [col for col in required_columns if col not in actual_columns]

assert not missing_required, (
    f"Missing required columns: {missing_required}. "
    "Please check the downloaded data."
)

print("Required column name check passed.")
print("Columns in data:", actual_columns)
```

```
Required column name check passed.
Columns in data: ['KEY', 'VENDOR_TYPE', 'STATUS', 'BUSINESS_NAME', 'LOCATION', 'DESCRIPTION', 'GEO_LOC
ALAREA', 'Geom', 'geo_point_2d']
```

Check 3: empty observations

For our modelling we only use the `BUSINESS_NAME` and `DESCRIPTION` columns. We do not expect any row where **both** of these fields are missing. If such rows existed, they would not be useful for our analysis, so we want to catch them early before splitting into train and test sets.

In [6]:

```
## Check 3: empty observations (rows with no info in key columns)

key_cols = ["BUSINESS_NAME", "DESCRIPTION"]

empty_rows_mask = food_vendors[key_cols].isna().all(axis=1)
n_empty_rows = empty_rows_mask.sum()

assert n_empty_rows == 0, (
    f"Found {n_empty_rows} rows where both BUSINESS_NAME and DESCRIPTION "
    "are missing. Please inspect and clean the raw data."
)

print(
    "Empty observation check passed: no rows with both key columns missing."
)
```

```
Empty observation check passed: no rows with both key columns missing.
```

Checks 4 and 5: Are the Data Types in each column the correct type and are the category levels correct?

In this check we will be using the pandera package to see if the relevant columns to our data analysis are the correct data types. Majority of the columns in our data frame will be dropped as they are not relevant to our project so we have assigned them to be `nullable=True` in the column constuctor meaning that we'll allow missing values in these unimportant columns. One of the crucial columns in our analysis is the `DESCRIPTION` column as this is the column that our target variable is based off of, so missing values in this column need to be flagged. The above check looks to make sure that **both** `BUSINESS_NAME` and `DESCRIPTION` arent misssing in the same row while check 4 makes sure that the `DESCRPTION` column in particular doesnt contain missing values.

To see if the catgory levels are correct in the `DESCRIPTION` column, the unique categories are listed and checked to make sure that no typos or spelling errors exist. With this we can see validate whether the values in our `DESCRITPION` column are correct or not using the pa.Check.isin() function from pandera.

In [7]:

```
## Checking the unique values in our DESCRIPTION column
unique_categories = food_vendors['DESCRIPTION'].unique()

## Listing out the possible values for DESCRIPTION
valid_categories_checked= [
'Hot Dogs', 'Vegetarian', 'French Crepes',
'Wraps and Sandwiches', 'Japanese Cuisine',
'Shawarma', 'Indian Cuisine','Italian Cuisine',
'Kosher Israeli', 'Juice and Smoothies','Fruit Shakes, Smoothies and Juices',
'Lemonade and Crepes','Sandwiches', 'Drinks and Smoothies',
'Middle Eastern Cuisine','Tacos and Burritos', 'Mexican Cuisine',
'Kebabs','Venezuelan Cuisine', 'Greek Cuisine',
'Comfort Food','Vegetarian and Vegan', 'Burgers and Fries',
'Australian Pies','Chinese Cuisine', 'Potato Based Dishes',
'Fried Chicken Sandwiches', 'Chowders and Soups', 'Dim Sum',
'Local Meats and Seafood', 'Asian Fusion', 'Thai Cuisine',
'Vegan Cuisine', 'Brasilian Cuisine', 'Variety Menu', 'Korean',
'Western', 'Grilled Cheese and Soups', 'Caribbean',
'Seafood and Western', 'International Comfort Foods',
'Tacos and Fresh Salsas', 'Central European Desserts']
```

In [8]:

```
## Check 4: correct data type?

#defining the schema
schema= pa.DataFrameSchema(
    {
        "KEY": pa.Column(str, nullable=True),
        "VENDOR_TYPE": pa.Column(str, nullable=True),
        "STATUS": pa.Column(str, nullable=True),
        "BUSINESS_NAME": pa.Column(str, nullable=True),
        "LOCATION": pa.Column(str, nullable=True),
        "DESCRIPTION": pa.Column(str, pa.Check.isin(valid_categories_checked)),
        "GEO_LOCALAREA": pa.Column(str, nullable=True),
        "Geom": pa.Column(object, nullable=True),
        "geo_point_2d": pa.Column(object, nullable=True),
    },
    drop_invalid_rows=True,
)
```

Checking our schema

If we run the schema.validate() method on our data set and the above test cases defined in the schema all pass, we should see our data frame. If not an error will be raised and we will be able to identify which column is failing the validity test. We will drop any invalid rows because in the instance a row in the `DESCRIPTION` column contains a missing value, we wont be able to generate our target class column so it's essentially useless. The same applies if a value in the `DESCRIPTION` column contains a typo or string mismatch.

In [9]:

```
schema.validate(food_vendors.head(), lazy=True)
```

Out[9]:

	KEY	VENDOR_TYPE	STATUS	BUSINESS_NAME	LOCATION	DESCRIPTION	GEO_LOCALAREA	
10	C1	vendor_food	open	Vie Niapolotan	Authorised Parking Meter - West Side of 400 Bu...	Italian Cuisine	Downtown	{"coordi [-123.1175335 49.286
73	C10	vendor_food	open	Mum's Grilled Cheese Truck	Authorised Parking Meter - West Side of 700 How...	Grilled Cheese and Soups	Downtown	{"coordi [-123.1206849 49.282
11	C11	vendor_food	open	Meet to Eat	Authorised Parking Meter South Side of 1000 Du...	Kosher Israeli	Downtown	{"coordi [-123. 49.285658],
1	C12	vendor_food	open	Chickpea	Authorised Parking Meter - North Side of W Cor...	Vegetarian	Downtown	{"coordi [-123.1157734 49.287
68	C13	vendor_food	open	Disco Cheetah	Authorised Parking Meter West Side of 700 Howe St	Korean	Downtown	{"coordi [-123.120443 49.2823

Table 2: Dataframe passing the validity test

Were any of our rows invalid?

We devised a handy code to see how many (if any) invalid rows were in our dataset! This is a simple yet effective way to observe the quality of the data that we are working with.

In [10]:

```python
before = len(food_vendors)
food_vendors_valid = schema.validate(food_vendors, lazy=True)
after = len(food_vendors_valid)
print(f"Dropped {before - after} invalid rows during data validation.")
```

```
Dropped 0 invalid rows during data validation.
```

What checks were not preformed?

When referring to the data validation checklist (https://ubc-dsci.github.io/reproducible-and-trustworthy-workflows-for-data-science/lectures/130-data-validation.html#data-validation-checklist) there are multiple checks that are not relevant/applicable to our project. We will briefly explain which checks are not covered in our project and why.

- Missingness not beyond expected threshold: In our analysis there are a few values in the `BUSINESS_NAME` column that are missing. This is alright in our project as one of the questions we are wanting to observe is whether or not a blank business name is relavent in predicitng our target variable (shown below). The majority of the columns aren't useful to us in our analysis so if they have null values we dont really care as

they'll be dropped in the preprocessing stage anyways and the one critical column (`DESCRIPTION`) has no leeway to have any missing values at all so if any missing values are present the row is dropped and a threshold isn't necessary.
- No duplicate observations for the combinations of `DESCRIPTION` and `BUSINESS_NAME` : This also doesn't apply to us as there is a possibility that the same resteraunt/food vendor exists in multiple locations so in this case the duplicates are important for us to see and include in our future analysis.

- No outlier or anomalous values: Doesn't apply to us as the relevant features in our model are text/strings so there's no way to test for outliers/anomolies

- Target/response variable follows expected distribution: In our project, there’s no real-world rule about how many hotdog vendors there should be compared to other food vendors, so the target variable doesn’t have an “expected” distribution. Because of this, it wouldn’t make sense to enforce a target-distribution check.

- No anomalous correlations between target/response variable and features/explanatory variables: This check is meant to catch features that are basically the target in disguise, or that are unrealistically predictive. In our project, we only use the business name as a feature, and we expect it to be related to whether a vendor sells hot dogs. We don’t have any extra columns that could secretly encode the target, so this kind of anomalous-correlation check isn’t applicable here.

- No anomalous correlations between features/explanatory variables: In our project, we only use one feature (`BUSINESS_NAME`), so there are no feature–feature pairs to compare. Because of this, a “no anomalous correlations between features” check does not meaningfully apply to our analysis.

Analysis

The motivation for our analysis is mainly understand the prediction power of the name of the vendor in identifying what they sell. In our case, we chose Hot Dogs, since in the preview of our data, we saw several instances of this category showing up.

Preparing the data for the analysis

Here we will be removing any column, except for the name and description. Then, we will add a category that states whether or not the vendors sell Hot Dogs or not.

In [11]:

```python
## dropping irrelevant columns
clean_food = food_vendors.drop(columns=[
    'KEY', 'VENDOR_TYPE', 'STATUS', 'LOCATION',
    'GEO_LOCALAREA', 'Geom', 'geo_point_2d'
])

## Here we are engineering the features for our models
clean_food["is_hotdog"] = clean_food["DESCRIPTION"] == "Hot Dogs"
clean_food["BUSINESS_NAME"] = clean_food["BUSINESS_NAME"].fillna("")

clean_food.head()
```

Out[11]:

	BUSINESS_NAME	DESCRIPTION	is_hotdog
10	Vie Niapolotan	Italian Cuisine	False
73	Mum's Grilled Cheese Truck	Grilled Cheese and Soups	False
11	Meet to Eat	Kosher Israeli	False
1	Chickpea	Vegetarian	False
68	Disco Cheetah	Korean	False

Table 3: Processed data from web source

With this, we can split our data into training and testing.

We chose a 70% split since our data set is small, and we do not want to risk underfitting on the test data.

We also set `random_state=522` to establish reproducibility for this analysis.

In [12]:

```
## create train and test split

train_data, test_data = train_test_split(
    clean_food, train_size=0.7, random_state=522
)
```

Exploratory Data Analysis (EDA)

Given that we are interested in solely the classification power of the name, we will be taking a look more into the classes present into our training set.

First, we will take a look at the DESCRIPTION category distribution:

In [13]:

```
## Data visualization for EDA
## Code in this cell adapted from DSCI 351 Lecture 2 and 5
## color names supported for the bar charts located at:
## https://www.w3schools.com/cssref/css_colors.php


plot1 = (alt.Chart(
    train_data,
    title=(
        "What are the most common cuisine types among food vendors "
        "in Downtown Vancouver?"
    )
).mark_bar
    (color="chocolate").encode(
        x=alt.X("count()", title="Total"),
        y=alt.Y("DESCRIPTION:N", sort='-x', title="Food type")
    ).properties(
        width=250,
        height=500
    )
)

plot1
```

Out[13]:

Fig. 1: Most common cuisine types among food vendors in Downtown Vancouver

As we can see, `Hot Dogs` are the most popular class in our data set, where can observe that the rest of our categories are pulverised into several other classifications with one or 2 observations. Thus, making our classifier identify the most popular one, might be a way to improve model performance.

Nonetheless, being the most popular class, does not mean that there could not be class imbalance. We will observe if this is true in the next plot:

In [14]:

```
plot2 = (alt.Chart(
    train_data,
    title="Are we dealing with a class imbalance in our train data?"
).mark_bar(color="seagreen").encode(
    x=alt.X("is_hotdog", title="Is it a hot dog vendor?"),
    y=alt.Y("count()", title="Number of vendors")
).properties(
        width=85,
        height=495
    )
        )

plot2
```

Out[14]:

Fig. 2: Class imbalance in training data

As we can see, even being the most popular class, there distribution is biased towards more non-hot-dog vendors (which we saw before that are not concentrated in a particular competing class). This implies there is a certain imbalance which should be noted when evaluating model performance.

Lastly, we noticed that there were some initial blanks in the `BUSINESS_NAME` which we addressed by changing it to an empty space (and thus not screwing up the `CountVectorizer` instance we will need for this analysis). It would be interesting to see if there is a discernible pattern of this, against the target:

In [15]:

```
# summary EDA, identify missing and NAN values

train_data["text_is_na"] = train_data["BUSINESS_NAME"] == ""

plot3 = (alt.Chart(
    train_data,
    title="Are blank names relevant for our classification?"
).mark_rect(color="seagreen").encode(
    x=alt.X("is_hotdog", title="Is it a hot dog vendor?"),
    y=alt.Y("text_is_na", title="Is it a blank BUSINESS_NAME?"),
    color=alt.Color("count()", title="# of observations")
).properties(
        width=200,
        height=200
    )
)

plot3 = plot3 + alt.Chart(train_data).mark_text(
    fontSize=14,
    fontWeight="bold"
).encode(
    x="is_hotdog:N",
    y="text_is_na:N",
    text=alt.Text("count():Q", format="d")
)

plot3
```

Out[15]:

Fig. 3: Blank names relevance in classification

We can observe here that all the cases of a blank `BUSINESS_NAME` are belong to Hot Dog vendors, which would be something we would like our models to capture.

In summary, when visualizing our EDA we can notice several key points. In the first plot we can see that of all the cuisine types from Downtown Vancouver food vendors, hot dog stands seem to be the most common of them all. It is also very important to analyze our classes before starting our work. When you have a large class imbalance, a lot of the times your model will give you a score that is not representative of whether or not your model works well. For example, if you observe the second plot for a data set where one class is represented in a much higher proportion than the other, a model like DummyClassifier will give you an extremely high score. This isn't because the model works perfectly it's because it'll always predict the higher represented class! Yet we can see that we do not have that issue as much here as the class imbalance isn't too skewed. Finally, we would expect for our classifier to be able to identify the "easy" base case of having no name, since this is a relevant discriminator for both our classes.

Methodology

For this analysis we will implement a `CountVectorizer` object to create a bag of words (BOW). This method will split each individual word in the names of the businesses into its own individual columns, and will assess whether or not the word is present in the data set.

We will then pass this `CountVectorizer` into a pipeline with the different models we want to test:

- `DummyClassifier` : This will be our baseline to check on how we will predict whether or not the vendor sells hot dogs or not.
- `DecisionTreeClassifier` : This simple model will help us identify if simple decisions map out the relationship of the names to the category.
- `LogisticRegression` : This model will help us identify if there are linear relationships in the model, and which tokens are more relevant for our classification.
- `BernoulliNB` (Naïve-Bayes): This model is quick to fit and train, and uses a probabilistic approach to the classification, and it would be interesting to see how it fares in comparison to the rest of them.

After performing the evaluation for each model, we will compare them all together and train the best one, optimising its hyperparameters. We will score based on the model **accuracy** (correct predictions over total predictions), as this is just a simple experiment to validate the relationship of these 2 variables.

With the best model, we will take a peek on how it performs on the test data, and evaluate our conclusions.

Helper Functions

These are some auxiliary methods to improve the readability of the analysis.

In [16]:

```
## This is to create the pipelines with the count vectorizer
## Note that we did not set "stop_words" as the filler words may
## be a relevant element of the title. Yet we will establish to
## only check for whether the word appears or not in the set.
## (binary=True)
def build_pipeline(model):
    return make_pipeline(
        CountVectorizer(binary=True),
        model
    )
```

```python
## Here we will store our model cross-validation (CV) results
model_comparison = dict()

## This is to perform the CV and store it for future comparison
def add_to_model_comparison(model_name, model):
    model_comparison[model_name] = pd.DataFrame(
        cross_validate(
            model,
            X_train,
            y_train,
            cv=5,
            return_train_score=True,
        )
    ).agg(['mean', 'std']).round(3).T

## This performs the CV  and displays scores
def show_cv_scores(model):
    return pd.DataFrame(
        cross_validate(
            model,
            X_train,
            y_train,
            cv=5,
            return_train_score=True,
        )
    )

## This performs the CV, stores the results and displays them
def record_and_display_cv_scores(model_name, model):
    cv_results = pd.DataFrame(
        cross_validate(
            model,
            X_train,
            y_train,
            cv=5,
            return_train_score=True,
        )
    )
    model_comparison[model_name] = cv_results.agg(
        ['mean', 'std']
    ).round(3).T
    return cv_results

## This concatenates the results and displays them in a formatted table
def compare_models(param='mean'):
    return pd.concat(
        model_comparison,
        axis='columns'
    ).xs(
        param,
        axis='columns',
        level=1
    ).style.format(
        precision=2
    ).background_gradient(
        axis=None
    )

## This displays the model mismatches for CV training or Test
## data after training
def display_model_mistmatches(model, train=True):
    if train:
        data_dict = {
            "y": y_train,
            "y_hat": cross_val_predict(
                model,
                X_train,
                y_train
            ).tolist(),
            "x": X_train.tolist(),
            "probabilities": cross_val_predict(
                model,
                X_train,
                y_train,
                method="predict_proba"
            ).tolist(),
        }
    else:
        data_dict = {
            "y": y_test,
            "y_hat": model.predict(X_test),
            "x": X_test.tolist(),
            "probabilities": model.predict_proba(X_test).tolist(),
        }

    df = pd.DataFrame(data_dict)
    return df[df["y"] != df["y_hat"]].sort_values('probabilities')


## This displays the confusion matrix for CV training
## or Test data after training
def display_confusion_matrix(model, train=True):
    if train:
        ConfusionMatrixDisplay.from_predictions(
            y_train,
            cross_val_predict(
                model,
                X_train,
                y_train
            )
        )
    else:
        ConfusionMatrixDisplay.from_predictions(
            y_test,
            model.predict(X_test)
        )
```

Data Split

We will perform the preparation of our data for the analysis, as well as getting some relevant features out of it.

In [17]:

```
## Here we are splitting our data into inputs and responses

X_train = train_data["BUSINESS_NAME"]
y_train = train_data["is_hotdog"]

X_test = test_data["BUSINESS_NAME"]
y_test = test_data["is_hotdog"]
```

One relevant element to observe is our vocabulary. We will extract how many words in total we can use, as well as some of them:

In [18]:

```
## Here we are obtaining the vocabulary of our BOW.

bag_of_words = make_pipeline(CountVectorizer(binary=True))
bag_of_words.fit(X_train, y_train)
vocab = (
    bag_of_words.named_steps["countvectorizer"].get_feature_names_out()
)

## Display the first 5 tokens

print("The vocabulary size is:", len(vocab))
pd.DataFrame({"words": vocab}).head()
```

```
The vocabulary size is: 95
```

Out[18]:

	words
0	actual
1	ali
2	arancino
3	arepa
4	arturo

Table 4: First 5 elements of the vocabulary.

We will also take a look at the proportions of our target variable in the training and test sets.

In [19]:

```
## look at the proportion of each class for train and test data

data_proportion = pd.DataFrame({
    "train": y_train.value_counts(normalize=True),
    "test": y_test.value_counts(normalize=True)
})

data_proportion
```

Out[19]:

	train	test
is_hotdog		
False	0.650794	0.678571
True	0.349206	0.321429

Table 5: Train and test proportions of the target class.

For all our models, a `True` prediction will mean that the vendor is a Hot Dog seller, and `False` the other way around.

As mentioned before, we can observe that the data is relatively balanced in both sets: being around **34%** of our total data. Yet the target class is less prominent in our test split. Nonetheless, we will not balance the data set, as this is something we can consider for a future iteration of this analysis.

Baseline (`DummyClassifier`)

Thus we will be defining a dummy classifier to have a reference to compare our different models. For this and all our models we will use 5 folds.

In [20]:

```
# DummyClassifier as baseline

dummy = DummyClassifier()

record_and_display_cv_scores("Baseline", dummy)
```

Out[20]:

	fit_time	score_time	test_score	train_score
0	0.001342	0.001150	0.615385	0.660000
1	0.000838	0.001041	0.615385	0.660000
2	0.000834	0.000879	0.692308	0.640000
3	0.000766	0.001888	0.666667	0.647059
4	0.000848	0.000888	0.666667	0.647059

Table 6: DummyClassifier cross validation scores and times.

As expected, the dummy consistently predicts the majority class, with accuracy of around **0.65**, being consistent with the representation of our split.

Decision Tree

Here we are training a simple decision tree to identify whether the vendor sells hot dogs or not. This is a simple model with easy to interpret coefficients, and it would be interesting checking whether or not it correctly identified some of the most relevant clues (something like "Joe's Hot Dogs" being correctly classified, for instance).

Cross Validation

In [21]:

```
tree = build_pipeline(DecisionTreeClassifier(random_state=522))

record_and_display_cv_scores("Decision Tree",  tree)
```

Out[21]:

	fit_time	score_time	test_score	train_score
0	0.004565	0.001480	0.615385	0.980000
1	0.002720	0.001502	0.692308	0.980000
2	0.002954	0.001280	0.692308	0.980000
3	0.002734	0.001133	0.416667	0.980392
4	0.002791	0.001075	0.416667	1.000000

Table 7: Decision tree cross validation scores and times.

The tree performs worse than the dummy classifier, as it is overfitting the prediction, which is evident in the substantial gap between the validation and training scores (around **-0.4** difference for all folds).

Model Parameters

We can take a look at the depths and tree structure to better understand these discrepancies:

In [22]:

```
tree.fit(X_train, y_train)

print(
    "The max depth of the tree is:",
    tree["decisiontreeclassifier"].tree_.max_depth
)
```

```
The max depth of the tree is: 34
```

As we can see, the model contains **34** levels of decisions, yet, the level of specificity (given that we are using a bag of words) makes it perform poorly. Here we can see the most discriminating factors:

In [23]:

```
plot_tree(
    tree["decisiontreeclassifier"],
    feature_names=vocab,
    max_depth=3,
    fontsize=7
)
plt.show()
```

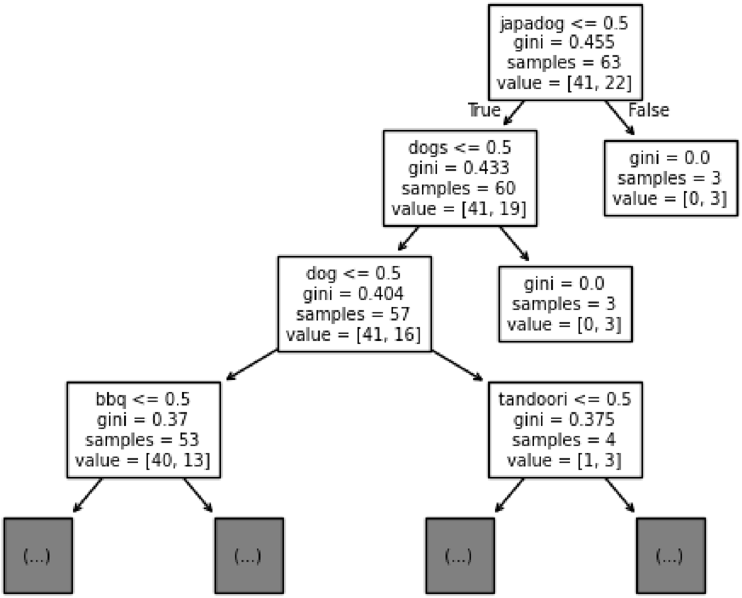

Fig. 4: Decision tree structure (limited to the first 5 levels of the depth)

We can observe some sensible initial discriminations, such as "dogs", "japadog" and "dog", which would quickly identify the vendor as a Hot Dog place.

Performance Metrics

Here we can observe the confussion metrics and misses in the cross validation of the model trained.

In [24]:

```
display_confusion_matrix(tree)
```

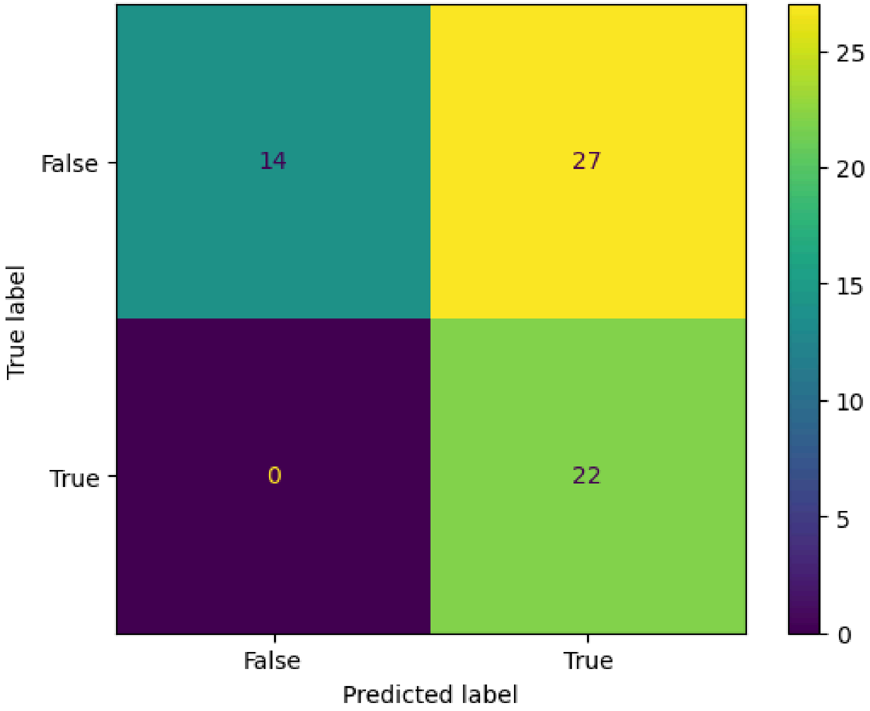

Fig. 5: Confusion matrix for the Decision Tree.

We can observe that the model is very good at identifying when something seems "Hot-Doggy", yet, it produces a high degree of false positives.

We can observe some of the mistakes below:

In [25]:

```
display_model_mistmatches(tree)
```

Out[25]:

	y	y_hat	x	probabilities
79	False	True	BBKB Kaboom Box	[0.0, 1.0]
29	False	True	Gourmet Syndicate	[0.0, 1.0]
60	False	True	Potato Tornado	[0.0, 1.0]
73	False	True	Mum's Grilled Cheese Truck	[0.0, 1.0]
40	False	True	Le Tigre	[0.0, 1.0]
88	False	True	Van Dog	[0.0, 1.0]
85	False	True	Suassy Thai	[0.0, 1.0]
51	False	True	Crab Park Chowdery	[0.0, 1.0]
19	False	True	Cheesey Does It	[0.0, 1.0]
14	False	True	San Juan Family Farm	[0.0, 1.0]
24	False	True	Actual Oden	[0.0, 1.0]
1	False	True	Chickpea	[0.0, 1.0]
6	False	True	Mogu	[0.0, 1.0]
35	False	True	Yokabai	[0.0, 1.0]
47	False	True	Marimba	[0.0, 1.0]
68	False	True	Disco Cheetah	[0.0, 1.0]
71	False	True	Master Chef's Kebab House	[0.0, 1.0]
61	False	True	Roaming Dragon	[0.0, 1.0]
9	False	True	Vij's Railway Express	[0.0, 1.0]
90	False	True	Commissary Connect	[0.0, 1.0]
82	False	True	La Bomba Taqueria	[0.0, 1.0]
41	False	True	Russett Shack	[0.0, 1.0]
25	False	True	Arturo's Mexico To Go	[0.0, 1.0]
2	False	True	Chou Chou Crepes	[0.0, 1.0]
27	False	True	Come Arepa	[0.0, 1.0]
81	False	True	Fat Duck Mobile Eatery	[0.0, 1.0]
22	False	True	Tacofino	[0.0, 1.0]

Table 8: Mismatches for Decision Tree.

Here we are seeing that the model tends to predict most of the time that the vendor is a hot dog stand, with probably the only reasonable exception being Van Dog.

Logistic Regression

Here we will train a logistic regression in order to see whether or not we can improve our accuracy and reduce the possible overfitting. This model also has the advantage of having interpretable parameters, which in our case relate how often each of our features is associated with the target variable (`is_hotdog = True`).

Cross Validation

In [26]:

```
lr = build_pipeline(LogisticRegression(random_state=522))

record_and_display_cv_scores("Logistic Regression", lr)
```

Out[26]:

	fit_time	score_time	test_score	train_score
0	0.010493	0.003414	0.692308	0.800000
1	0.009657	0.001519	0.769231	0.820000
2	0.008728	0.003329	0.769231	0.820000
3	0.007724	0.003158	0.916667	0.784314
4	0.006582	0.001161	0.750000	0.823529

Table 9: Logistic regression cross validation scores and times.

We can see a slight improvement in generalisation from the decision tree, but it performs not much better than the dummy regressor. This could indicate that there may not be a single independent linear relationship in the token features to the target. Also it is worth noting that the model has a lot variability between folds.

Model Parameters

We can take a look at the coefficients associated with each word and see which ones are the most relevant:

In [27]:

```
lr.fit(X_train, y_train)

print(
    "Number of coefficients: ",
    len(lr["logisticregression"].coef_[0]),
)
```

```
Number of coefficients:  95
```

In [28]:

```
lr_coefficients = pd.DataFrame({
    "token": vocab,
    "coefficient": lr["logisticregression"].coef_[0]
})
print(
    "The intercept is:",
    lr["logisticregression"].intercept_[0]
)
lr_coefficients
```

```
The intercept is: -0.3346403800360456
```

Out[28]:

	token	coefficient
0	actual	-0.287395
1	ali	0.258929
2	arancino	-0.209049
3	arepa	-0.287395
4	arturo	-0.195682
...	...	...
90	truck	-0.463946
91	van	0.208222
92	vij	-0.251701
93	wraps	-0.330149
94	yokabai	-0.337719

95 rows × 2 columns

Table 10: Coefficients of the logistic regression.

In [29]:

```
visualize_coefficients(
    lr_coefficients['coefficient'].to_numpy(),
    vocab,
    n_top_features=5
)
```

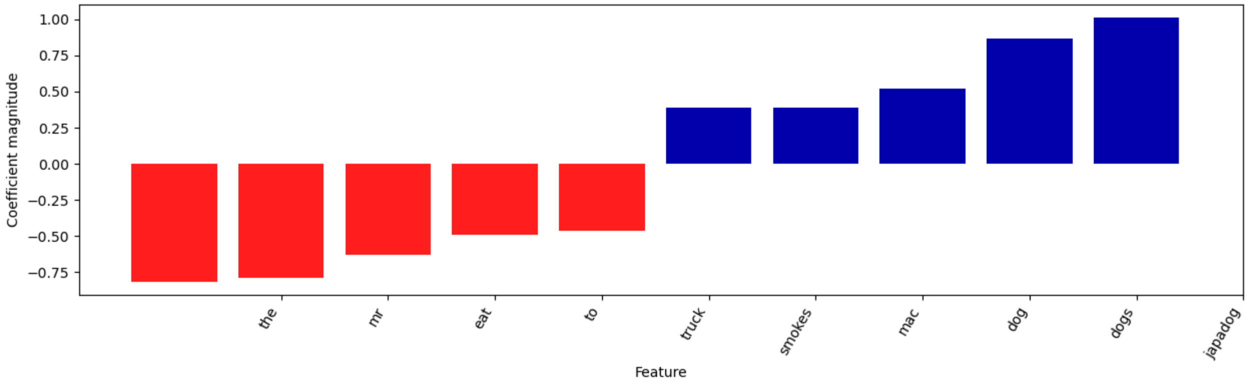

Fig. 6: Top 5 most discriminant features (upper and lower).

We can observe that the coefficients associated roughly match with the choices determined by the decision tree, with "japadog", "dog", and "dogs" being relevant.

Yet if we observe the intercept, we see the model is heavily biased into making a negative prediction. Thus, we would not expect the model being good in identifying hot dog places in particular.

Performance Metrics

Here we can observe the confusion metrics and misses in the cross validation of the model.

In [30]:

```
display_confusion_matrix(lr)
```

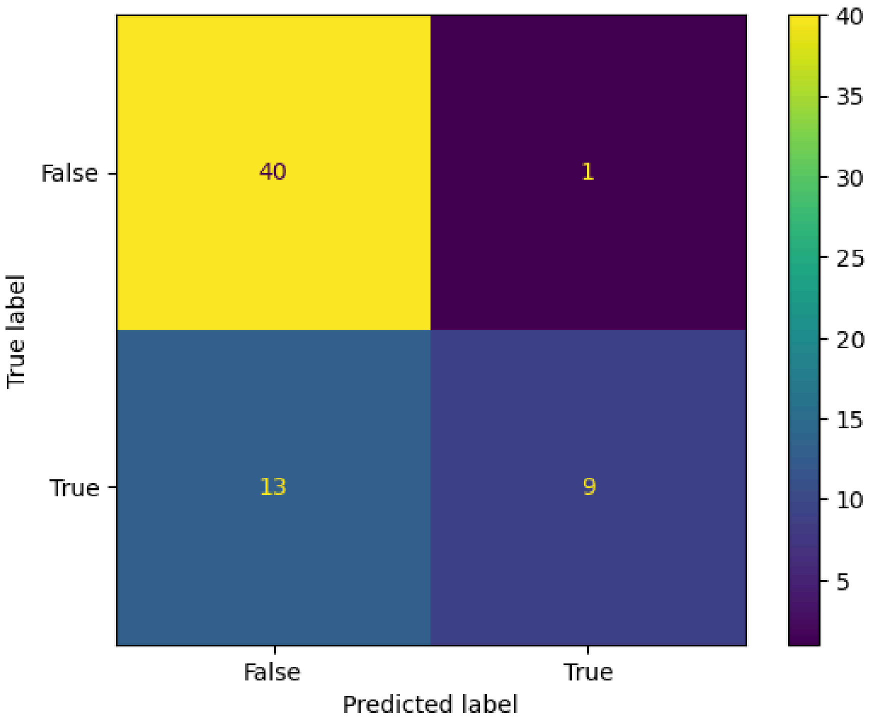

Fig. 7: Confusion matrix for the Logistic Regression.

We can observe that the logistic regression is not particularly good at discriminating, since it is evidently favouring the "not hot dog" class, as we expected from the coefficients calculated.

We can also see the patterns for the mismatches:

In [31]:

```
display_model_mismatches(lr)
```

Out[31]:

	y	y_hat	x	probabilities
88	False	True	Van Dog	[0.48486372091146257, 0.5151362790885374]
0	True	False		[0.5575291620674792, 0.4424708379325208]
23	True	False		[0.5665436668638206, 0.43345633313617943]
78	True	False		[0.5665436668638206, 0.43345633313617943]
44	True	False		[0.5951209415798449, 0.4048790584201552]
77	True	False		[0.5951209415798449, 0.4048790584201552]
37	True	False		[0.5951209415798449, 0.4048790584201552]
64	True	False		[0.6011852449918378, 0.39881475500816216]
16	True	False		[0.6011852449918378, 0.39881475500816216]
45	True	False	Holy Smokes	[0.6011852449918378, 0.39881475500816216]
46	True	False	Mac BBQ	[0.6011852449918378, 0.39881475500816216]
53	True	False		[0.6048256520648894, 0.39517434793511064]
76	True	False		[0.6048256520648894, 0.39517434793511064]
89	True	False		[0.6048256520648894, 0.39517434793511064]

Table 11: Mismatches for Logistic Regression.

We can see the model failed identifying some of the "easy" catches we found previously, such as identifying blanks or keywords like "dog" which do not skew the balance enough in favour of the target class.

Naïve-Bayes

Finally we will be testing the Naïve-Bayes model, which is also a relatively simple model that also does not tend to over-fit as much, just to see which model is best.

Cross Validation

In [32]:

```
naive_bayes = build_pipeline(BernoulliNB())

record_and_display_cv_scores("Naïve-Bayes", naive_bayes)
```

Out[32]:

	fit_time	score_time	test_score	train_score
0	0.004410	0.001711	0.692308	0.800000
1	0.005544	0.002494	0.769231	0.820000
2	0.002577	0.001611	0.769231	0.820000
3	0.004035	0.001853	0.916667	0.784314
4	0.002347	0.001194	0.750000	0.823529

Table 12: Naïve-Bayes cross validation scores and times.

We can also see that it performs better than the tree and dummy with less overfit, although not very consistently, like the logistic regression.

Performance Metrics

Here we can observe the confusion metrics and misses in the cross validation of the model trained.

In [33]:

```
display_confusion_matrix(naive_bayes)
```

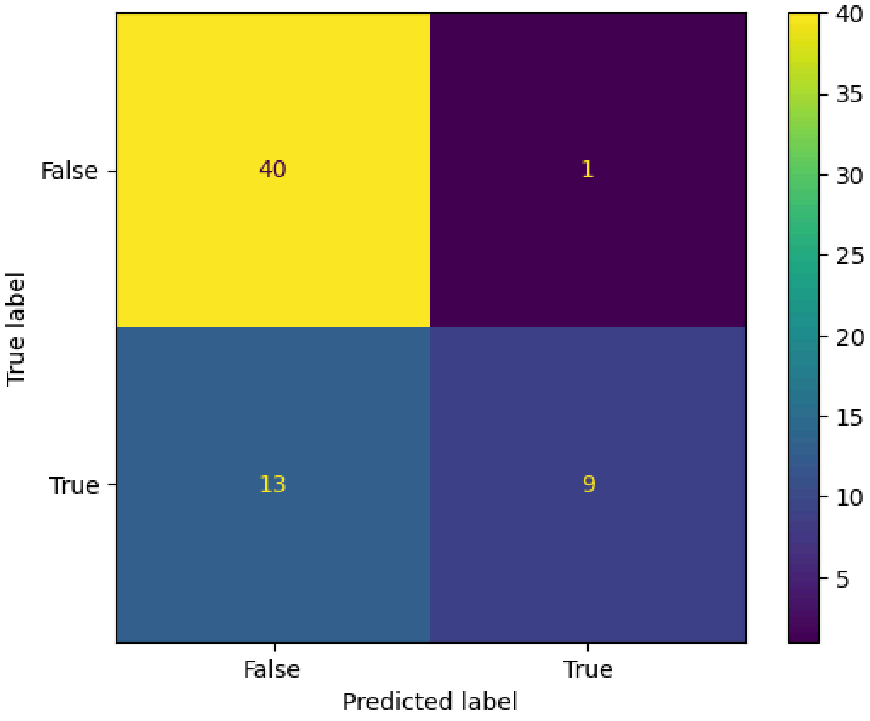

Fig. 8: Confusion matrix for the Naïve-Bayes classifier

It performs quite similar to the logistic regression, as we can see is not particularly good at identifying hot dog features.

We can also see its mismatches:

In [34]:

```
display_model_mismatches(naive_bayes)
```

Out[34]:

	y	y_hat	x	probabilities
88	False	True	Van Dog	[0.36422282298952524, 0.6357771770104752]
0	True	False		[0.6544133650223363, 0.34558663497766406]
23	True	False		[0.6839649670539567, 0.3160350329460436]
78	True	False		[0.6839649670539567, 0.3160350329460436]
44	True	False		[0.7066548087250116, 0.29334519127498854]
77	True	False		[0.7066548087250116, 0.29334519127498854]
37	True	False		[0.7066548087250116, 0.29334519127498854]
64	True	False		[0.7247833230416214, 0.27521667695837865]
16	True	False		[0.7247833230416214, 0.27521667695837865]
45	True	False	Holy Smokes	[0.7247833230416214, 0.27521667695837865]
46	True	False	Mac BBQ	[0.7247833230416214, 0.27521667695837865]
53	True	False		[0.7834187709904163, 0.21658122900958396]
76	True	False		[0.7834187709904163, 0.21658122900958396]
89	True	False		[0.7834187709904163, 0.21658122900958396]

Table 13: Mismatches for Naïve-Bayes.

We see a similar pattern as the logistic regression, failing to identify the "obvious" patterns we stated in the beginning.

Model Comparisons

Knowing this, we can compare their scores to determine the model to train.

In [35]:

```
compare_models()
```

Out[35]:

	Baseline	Decision Tree	Logistic Regression	Naïve-Bayes
fit_time	0.00	0.00	0.01	0.00
score_time	0.00	0.00	0.00	0.00
test_score	0.65	0.57	0.78	0.78
train_score	0.65	0.98	0.81	0.81

Table 14: Comparisons of mean values of scores and times for all models.

We can see a negligible difference between Naïve-Bayes and LR. Yet, to be able to proceed, we will proceed train Naïve-Bayes classifier, as it is also a relatively simple model with slighlty faster fitting time.

Best Model Hyperparameter Optimisation

Here we will perform the optimisation of our model. Given that we chose the Naïve-Bayes estimator, we will optimise the `alpha` hyperparameter (which controls the tradeoff between variance and bias of our model), as well as the `max_features` variable (the actual size of our vocabulary considering the top `max_features` words) of our `CountVectorizer`, as this can also play a role in overfitting.

We are using a randomised approach to test in a wide space, with 500 iterations and a random integer ranging from $[5, \text{size of the vocabulary}]$ for `max_features`, and a loguniform distribution ranging from $[0.001, 1000]$ for `alpha`.

After this, we will use the best parameters obtained, and train our best model.

In [36]:

```
param_grid = {
    "countvectorizer__max_features": randint(5, len(vocab)),
    "bernoullinb__alpha": loguniform(1e-2, 1e2),
}

random_search = RandomizedSearchCV(
    naive_bayes,
    param_distributions=param_grid,
    n_iter=500,
    n_jobs=-1,
    return_train_score=True,
)

random_search.fit(X_train, y_train)

print(
    "Random Search best model score: \t %0.3f"
    % random_search.best_score_
)
print(
    "Random Search best max_features: \t %0.3f"
    % random_search.best_params_["countvectorizer__max_features"]
)
print(
    "Random Search best alpha: \t\t %0.3f"
    % random_search.best_params_["bernoullinb__alpha"]
)

pd.DataFrame(random_search.cv_results_)[
    [
        "mean_test_score",
        "param_countvectorizer__max_features",
        "param_bernoullinb__alpha",
        "mean_fit_time",
        "rank_test_score",
    ]
].set_index("rank_test_score").sort_index().head()
```

```
Random Search best model score:          0.779
Random Search best max_features:         91.000
Random Search best alpha:                0.738
```

Out[36]:

	mean_test_score	param_countvectorizer__max_features	param_bernoullinb__alpha	mean_fit_time
rank_test_score				
1	0.779487	91	0.737887	0.008565
1	0.779487	70	0.775228	0.003835
1	0.779487	49	0.819482	0.003113
1	0.779487	62	0.683604	0.005210
1	0.779487	74	0.782083	0.004063

Table 15: Best hyperparameters for the best model (Logistic Regression).

We can observe a slight increase in the validation score, but in order to prevent overfitting on the validation set, we will compare our model with the actual test data:

In [37]:

```
print(
    "Best model test score:",
    f'{random_search.score(X_test, y_test):.3f}'
)
```

```
Best model test score: 0.714
```

The model performs quite similar to the validation score, which likely tells us that it was able to generalise and learn.

In order to validate this assumption, we can take a look at the confusion matrix

In [38]:

```
## Confusion matrix for test predictions

display_confusion_matrix(random_search.best_estimator_, train=False)
```

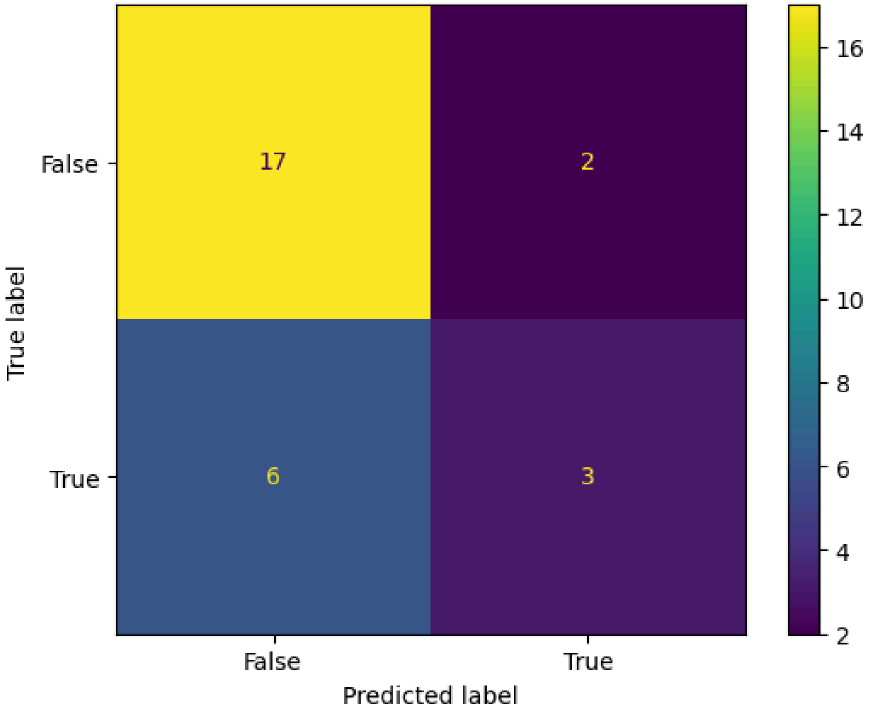

Fig. 9: Confusion matrix for the best classifier

We can see that, although the model makes good predictions, it still fails to raise the "hot dog alert".

We can further see this in the failed classifications:

In [39]:

```
display_model_mistmatches(random_search.best_estimator_, train=False)
```

Out[39]:

	y	y_hat	x	probabilities
72	False	True	Mo's Hot Dog Plus	[0.07439273399763462, 0.9256072660023648]
17	False	True	Lemon Sea	[0.262963390992561, 0.7370366090074388]
38	True	False		[0.6173615782462664, 0.3826384217537337]
50	True	False		[0.6173615782462664, 0.3826384217537337]
15	True	False		[0.6173615782462664, 0.3826384217537337]
52	True	False		[0.6173615782462664, 0.3826384217537337]
36	True	False		[0.6173615782462664, 0.3826384217537337]
13	True	False	Mr. Tube Steak	[0.8619151163248293, 0.13808488367517086]

Table 16: Mismatches for best model.

Save for the first one, which we would even classify as a hot dog stand, it still missed some of the cues we identified at the beginning, meaning this model will probably will encounter this limitations in future predictions.

Discussion

We compared the performance of 4 different models (Dummy, Decision Tree, Logistic Regression, and Naive-Bayes), and we proceeded with Naive-Bayes, therefore, we chose Naive-Bayes to move forward with hyperparameter tuning. We used RandomizedSearchCV to identify the best hyperparameter values for Max Features and alpha, and evaluated the best model with the test set.

As we saw, there are several limitations to what a count vectorizer and a binary classification can perform. The relationships that we found within our variables are probably not linear, as there are some cases where the biases of our more intelligent classifiers, such as the Bayesian and the logistic regression, would favor into classifying something as not hotdog, when we noticed from our EDA that it was those specific cases of no name where the model should have predicted that that was hotdog stand. This makes for a model that will be particularly good at identifying the majority class, which pretty much makes it comparable to a dummy. And thus, we observed the limitations of of our current estimators.

Now, depending on the context of our problem, we may lean in favour of having a model that is really good at predicting when something is not a hotdog, versus wanting a model that is really good at predicting when something is a hotdog.

Let's say we have someone who doesn't really like hot dogs that much. We would prefer a model that probably outputs more consistently or classifies non-hot dog places as non-hot dog places where this default probably like opens up possibilities for someone looking for options that are likely not hot dog related. And it is not too terrible if hot dog place slides in given that it is the fewer of the bunch.

For that particular case, our model probably is the one fitting better into that narrative as it is consistent enough to determining or correctly classifying the null class even though if it's not as good as classifying the positive class.

Now, in the context of someone really craving a hot dog and wanting to be very sure that that is a hot dog place, then probably the best model that we trained will not fit into that description as much, since it's not particularly good into predicting a class. For that case, it would have been better to train probably a decision tree, which we saw had much higher bias into identifying the hot dog class.

A test accuracy of 0.71 shows that our model is still a work in progress, but it shows promising results from the confusion matrix, where it only has 1 FP. There are also 6 FN, as that is a byproduct of how the model learned the patterns.

Some challenges in the data set are the size. It is a small dataset with only ~90 entries. Another challenge is the imbalance of classes. It would be ideal if each class would represent roughly 50–50% of the samples. We believe that the imbalance was not severe, so we didn’t make any adjustments. In the future, we can take the argument ”class_weight” into account during hyperparameter tuning for a model that supports it.

Finally, we could also add for future iterations, or as a different research question, whether an SVM would perform better given the conditions we have mentioned. It is likely that a nonlinear model will likely fare better since we have found some instances where the natural bias of logistic regression and Naïve-Bayes have pushed the model to incorrectly classify some of the examples.

References

- UBC Master of Data Science Program. DSCI 531: Effective use of Visual Channels– Lecture 2: Bar Chart syntax. 2025.
- UBC Master of Data Science Program. DSCI 531: Visualization for communication– Lecture 5: axis label formatting. 2025.
- W3Schools. CSS Color Names. W3Schools.com. https://www.w3schools.com/cssref/css_colors.php (accessed 21 November 2025).
- UBC Master of Data Science Program. DSCI 571: Lecture 8: Linear Models. 2025.

- UBC Master of Data Science Program. DSCI 562: Data Validation in Statistical Workflows – Lecture 16: Data Validation Checklist. 2025