CARTE-Enbridge Bootcamp

AI in Market Strategy

We are starting off today a little differently! Because the value of AI in Market Strategy is centred around specific applications, we are going to work through three different case studies. Each case study will focus on both a different domain and a different technology. By the end, we will have a strong understanding of the growing role of AI in Market Strategy!

Case Study 1: Predictive Analytics

To begin with, we will be looking at a dataset of avocado prices and demand over a three-year period. Grocery stores need to understand trends in demand and pricing for avocados, to ensure they have enough stock and to ensure they are pricing their avocados competitively. We will be using a toolkit from Meta (aka Facebook) called Prophet. Prophet is a forecasting tool that is designed to be easy to use, and to produce forecasts that are both accurate and explainable.

Load the dataset in the cell below. Because we are using time-series data, we instruct Pandas to `parse` the dates in the dataset. This allows us to do things like compute the time between two dates, or to group data by year, month, or day. We specify the format to be `YYYY-MM-DD`, which is represented by `%Y-%m-%d`.

In [1]:
```
import pandas as pd

df = pd.read_csv("https://github.com/alexwolson/carte_workshop_datasets/raw/main/avocado.csv.zip", compression="zip", inde
df["Date"] = pd.to_datetime(df["Date"], format="%Y-%m-%d")
df.set_index("Date", inplace=True)
```

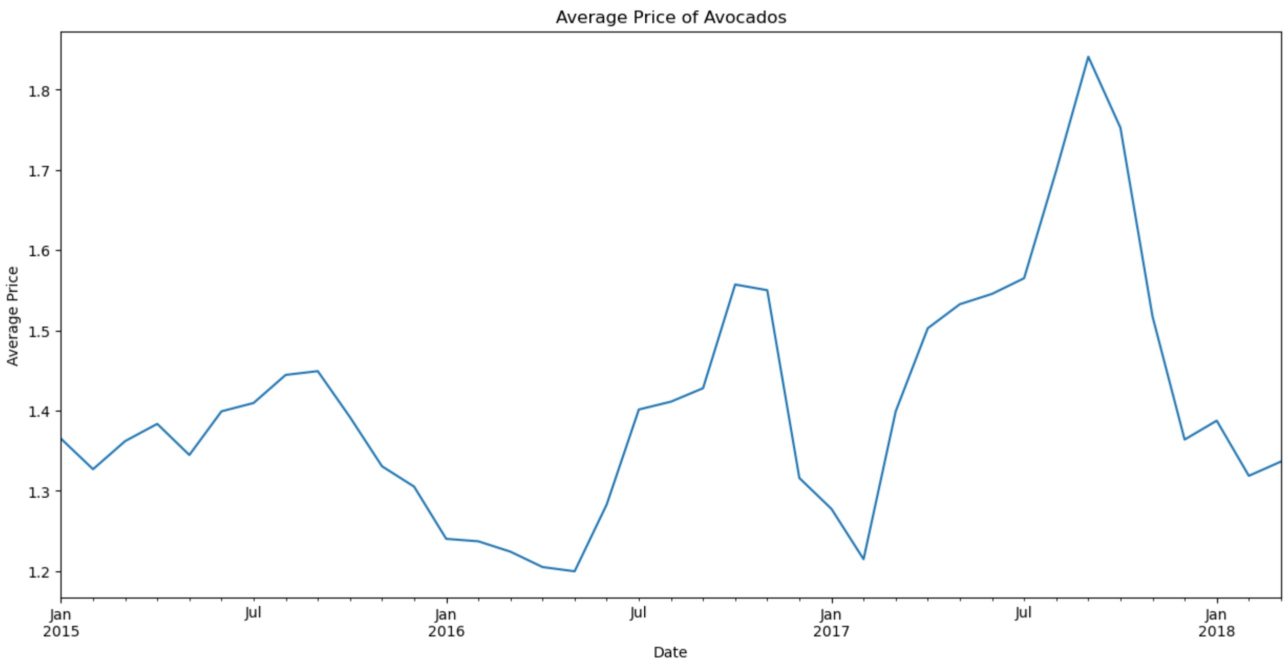
In [2]:
```
df.head() # 4046, 4225, 4770 are the PLU codes for different types of avocados
```

Out[2]:

Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarge Bags	type	year	region
2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0.0	conventional	2015	Albany
2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0.0	conventional	2015	Albany
2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14	0.0	conventional	2015	Albany
2015-12-06	1.08	78992.15	1132.00	71976.41	72.58	5811.16	5677.40	133.76	0.0	conventional	2015	Albany
2015-11-29	1.28	51039.60	941.48	43838.39	75.78	6183.95	5986.26	197.69	0.0	conventional	2015	Albany

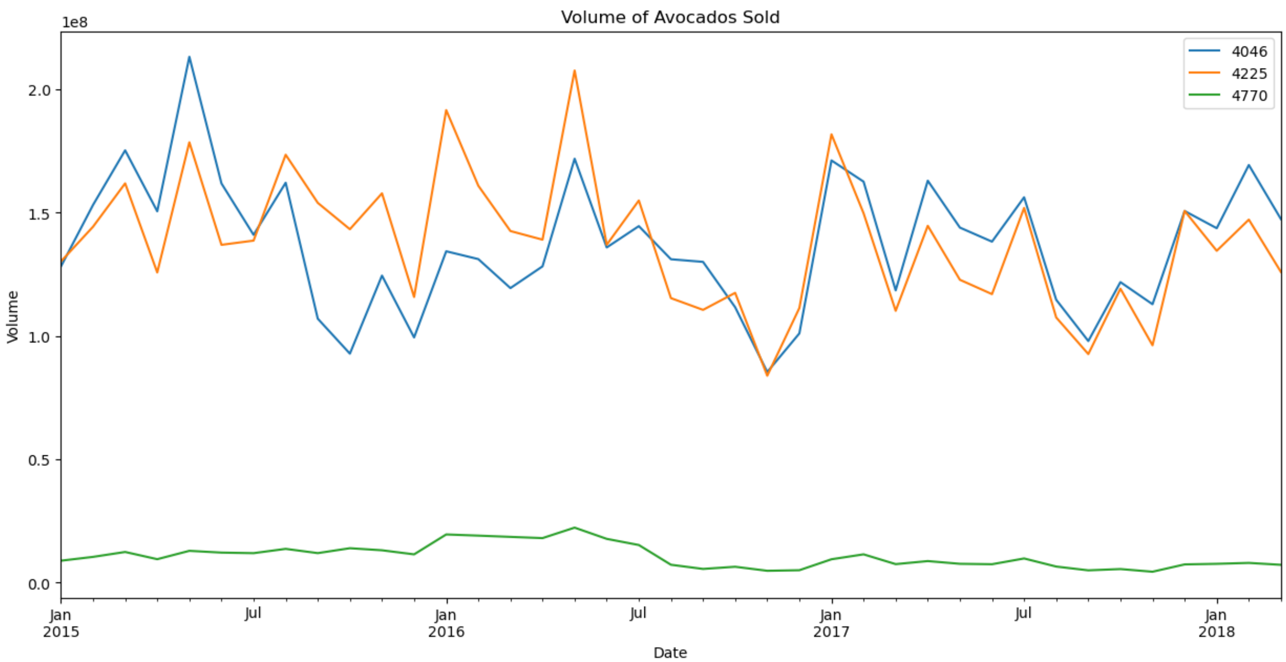
As ever, we will start by exploring the data. Let's plot the average price of avocados over time. We can use the `resample` method to group the data by month, and then take the average of each group. We can then plot the result using the `plot` method.

In [3]:
```
import matplotlib.pyplot as plt

df.resample("M")["AveragePrice"].mean().plot(figsize=(15,7))
plt.ylabel("Average Price")
plt.title("Average Price of Avocados")
plt.show()
```



Let's also look at the volume of each PLU code sold over time:

In [4]:
```
df.resample("M")[["4046", "4225", "4770"]].sum().plot(figsize=(15,7))
plt.ylabel("Volume")
plt.title("Volume of Avocados Sold")
plt.show()
```



Now let's move to building a predictive model. We will use Prophet to predict the average volume of avocados sold. Prophet is designed to be easy to use, and to produce forecasts that are both accurate and explainable. We will start by creating a new DataFrame with the columns that Prophet expects: `ds` for the date, and `y` for the value we want to predict. Since we want to be able to evaluate the quality of our predictions, we will separate out the last 6 months of data as a test set.

In [5]:
```
prophet_df = df[["Total Volume"]].resample("W").sum().reset_index() # Aggregate to the week level
prophet_df.columns = ["ds", "y"]
prophet_df_train = prophet_df[:-26] # All but the last six months
prophet_df_test = prophet_df[-26:]
```

Now we can create a Prophet model and fit it to our training data. Prophet supports automatically considering holidays, but we don't expect holidays to have a large impact on avocado sales, so we won't take advantage of this. In other contexts, considering things like weather, 'shocks' (e.g. a pandemic), or other events can be very important.

In [6]:
```
!pip install -U -q prophet plotly fastapi kaleido python-multipart uvicorn "typing-extensions<4.6.0"
```

In [7]:
```
from prophet import Prophet
from time import time

model = Prophet(interval_width=1)
start_time = time()
model.fit(prophet_df_train)
print(f'Training time: {time() - start_time} seconds')
```

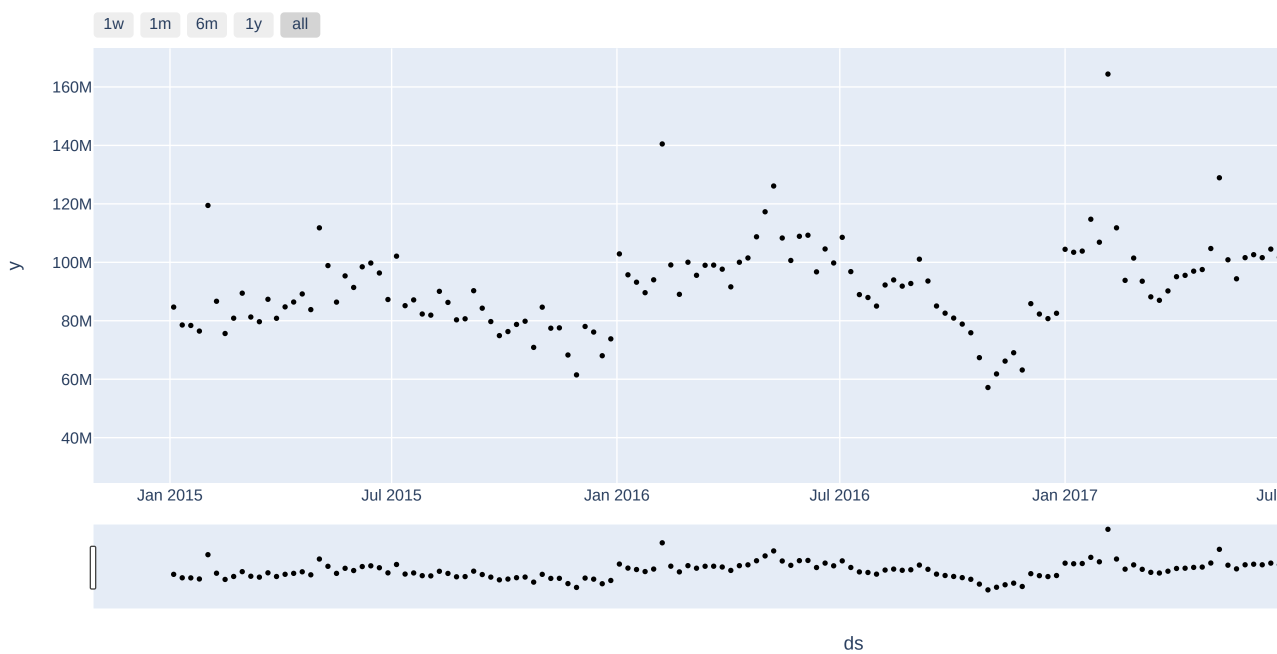
```
16:43:58 - cmdstanpy - INFO - Chain [1] start processing
16:43:58 - cmdstanpy - INFO - Chain [1] done processing
Training time: 0.029536962509155273 seconds
```

In [8]:
```
predictions = model.predict(prophet_df_test)
# Calculate percentage of true values that fall between yhat_lower and yhat_upper
correct = []
for i in range(len(predictions)):
    if (prophet_df_test["y"].iloc[i] >= predictions["yhat_lower"].iloc[i]) and (prophet_df_test["y"].iloc[i] <= predictior
        correct.append(1)
    else:
        correct.append(0)
print(f"Percentage of true values that fall between yhat_lower and yhat_upper: {sum(correct)/len(correct) * 100:.2f}%")
```

```
Percentage of true values that fall between yhat_lower and yhat_upper: 84.62%
```

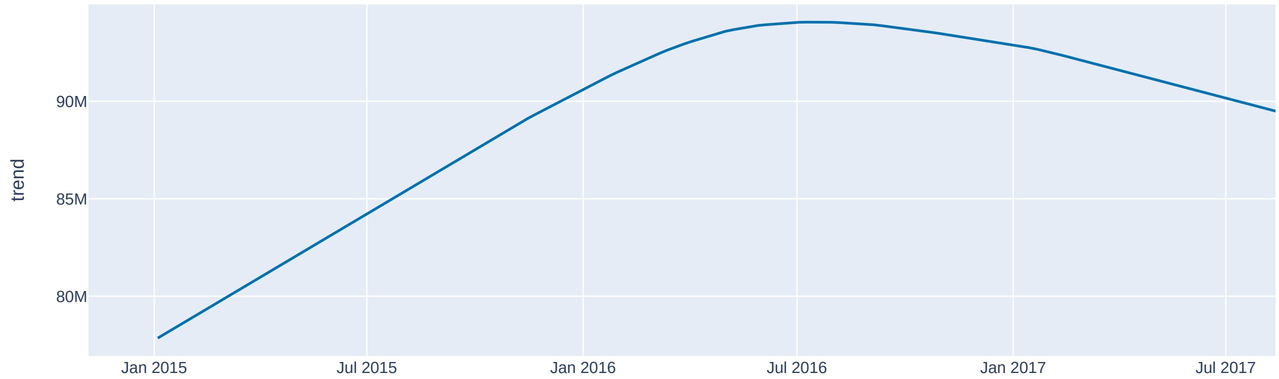
We can see that our model is able to predict the volume of avocados sold with a reasonable degree of accuracy. We can visualize the predictions using the `plot` method. The black dots represent the actual values, and the blue line represents the predictions. The shaded blue area represents the uncertainty in the predictions.

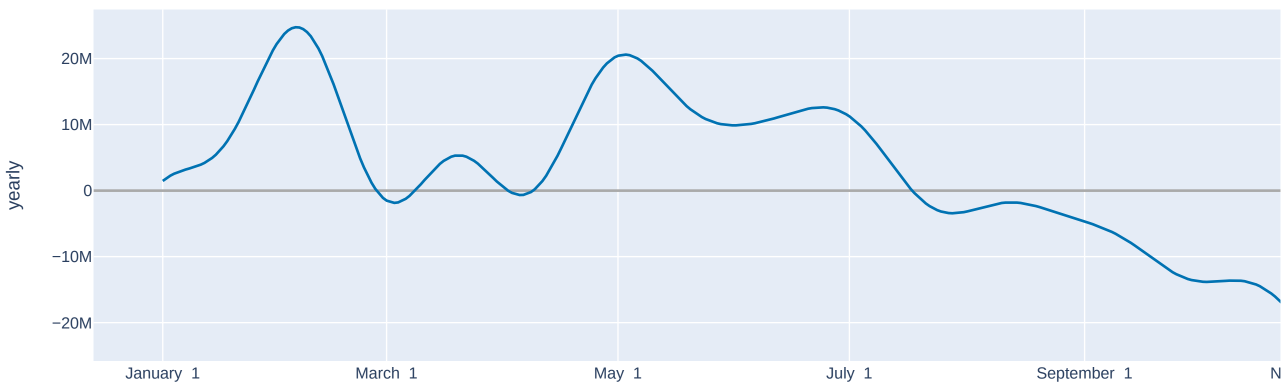
In [9]:
```
from prophet.plot import plot_plotly, plot_components_plotly

fig = plot_plotly(model, model.predict(prophet_df_test), figsize=(1300,600))
fig.show()
```



We can also break down the predictions into their components. The first plot shows the overall trend in avocado sales, and the second plot shows the weekly seasonality.

In [10]:
```
fig = plot_components_plotly(model, model.predict(prophet_df), figsize=(1300,400))
fig.show()
```





Your turn

The avocado dataset breaks down the data by organic and conventional avocados. Using the separated datasets below, fit two distinct models to predict the volume of organic and conventional avocados sold. How do the predictions compare? What are the main differences between the two models?

In [11]:
```
prophet_df_conventional = df[df["type"] == "conventional"][["Total Volume"]].resample("W").sum().reset_index()
prophet_df_conventional.columns = ["ds", "y"]
prophet_df_conventional_train = prophet_df_conventional[:-26]
prophet_df_conventional_test = prophet_df_conventional[-26:]

prophet_df_organic = df[df["type"] == "organic"][["Total Volume"]].resample("W").sum().reset_index()
prophet_df_organic.columns = ["ds", "y"]
prophet_df_organic_train = prophet_df_organic[:-26]
prophet_df_organic_test = prophet_df_organic[-26:]
```

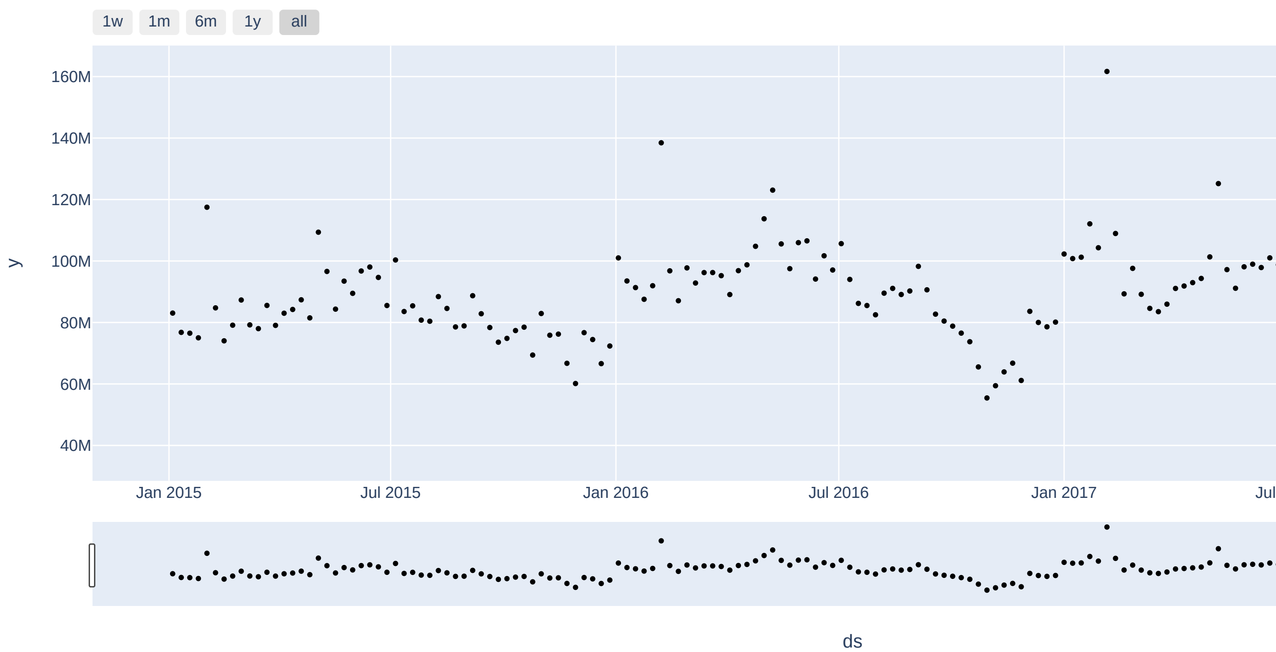
In [12]:
```
model_conventional = Prophet(interval_width=1)
start_time = time()
model_conventional.fit(prophet_df_conventional_train)
print(f'Training time: {time() - start_time} seconds')

model_organic = Prophet(interval_width=1)
start_time = time()
model_organic.fit(prophet_df_organic_train)
print(f'Training time: {time() - start_time} seconds')
```

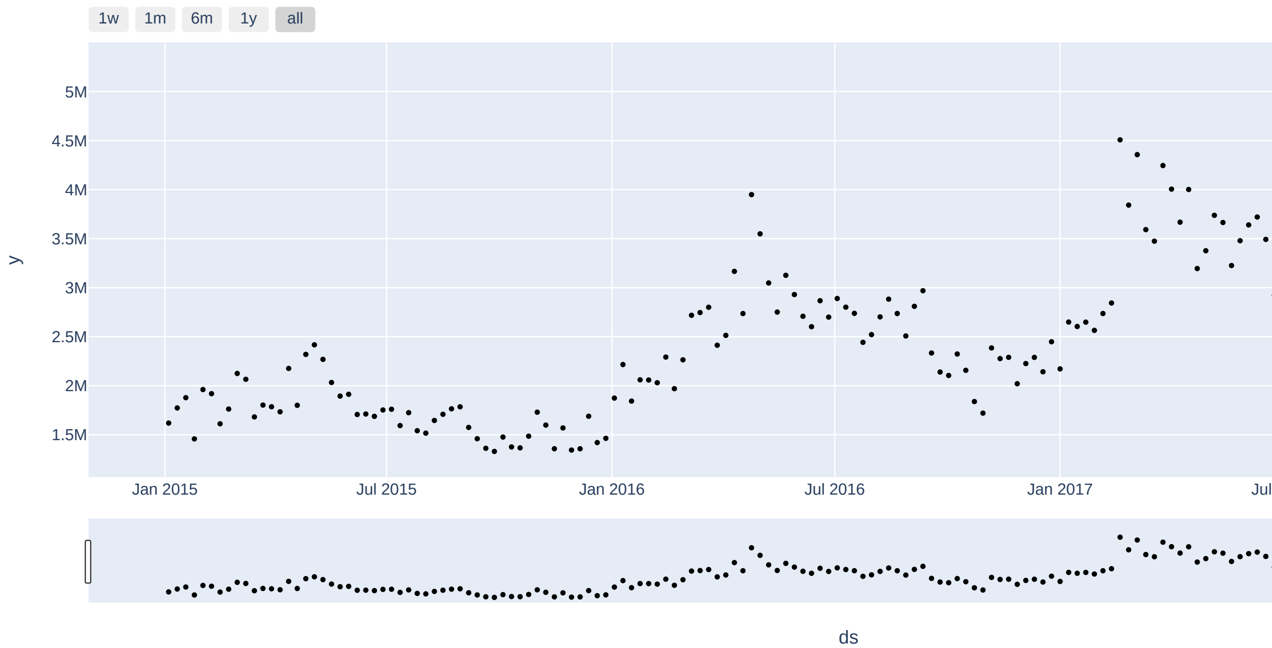
```
16:43:58 - cmdstanpy - INFO - Chain [1] start processing
16:43:58 - cmdstanpy - INFO - Chain [1] done processing
16:43:58 - cmdstanpy - INFO - Chain [1] start processing
16:43:58 - cmdstanpy - INFO - Chain [1] done processing
Training time: 0.026979446411132812 seconds
Training time: 0.030364274978637695 seconds
```

In [13]:
```
predictions_conventional = model_conventional.predict(prophet_df_conventional_test)
predictions_organic = model_organic.predict(prophet_df_organic_test)
```

```
correct_conventional = []
for i in range(len(predictions_conventional)):
    if (prophet_df_conventional_test["y"].iloc[i] >= predictions_conventional["yhat_lower"].iloc[i]) and (prophet_df_conve
        correct_conventional.append(1)
    else:
        correct_conventional.append(0)

correct_organic = []
for i in range(len(predictions_organic)):
    if (prophet_df_organic_test["y"].iloc[i] >= predictions_organic["yhat_lower"].iloc[i]) and (prophet_df_organic_test["y
        correct_organic.append(1)
    else:
        correct_organic.append(0)

print(f"Percentage of true values that fall between yhat_lower and yhat_upper for conventional: {sum(correct_conventional)
print(f"Percentage of true values that fall between yhat_lower and yhat_upper for organic: {sum(correct_organic)/len(corre

fig = plot_plotly(model_conventional, model_conventional.predict(prophet_df_conventional_test), figsize=(1300,600))
fig.show()

fig = plot_plotly(model_organic, model_organic.predict(prophet_df_organic_test), figsize=(1300,600))
fig.show()
```

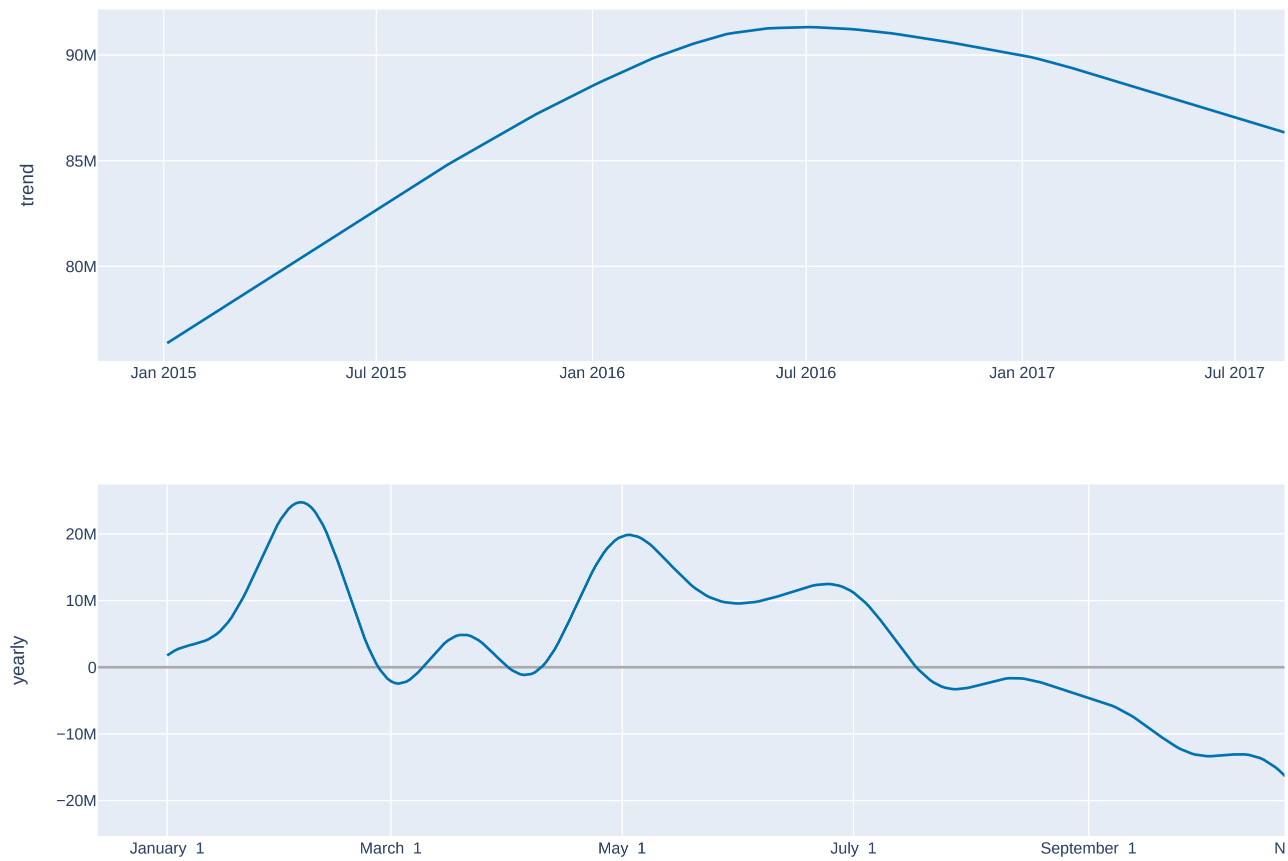
```
Percentage of true values that fall between yhat_lower and yhat_upper for conventional: 88.46%
Percentage of true values that fall between yhat_lower and yhat_upper for organic: 96.15%
```



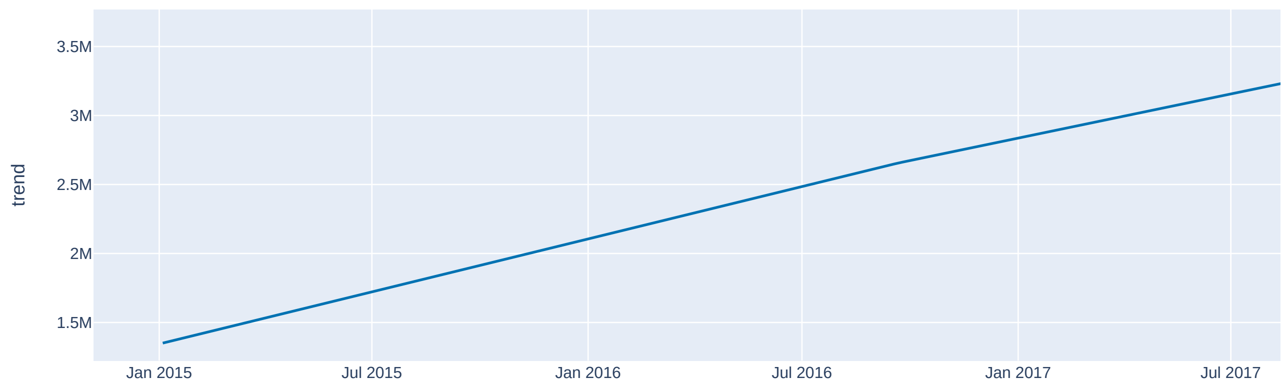


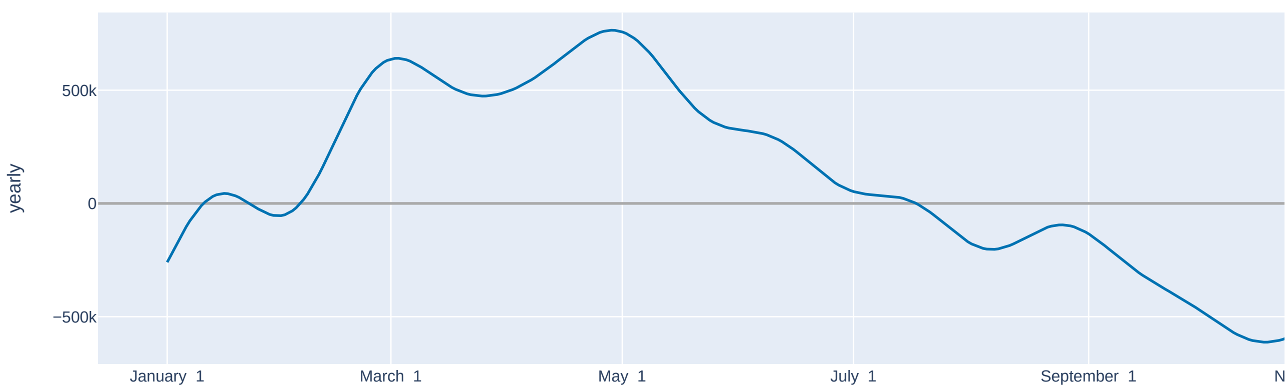
In [14]:

```
fig = plot_components_plotly(model_conventional, model_conventional.predict(prophet_df_conventional), figsize=(1300,400))
fig.show()

fig = plot_components_plotly(model_organic, model_organic.predict(prophet_df_organic), figsize=(1300,400))
fig.show()
```



trend
90M
85M
80M
Jan 2015
Jul 2015
Jan 2016
Jul 2016
Jan 2017
Jul 2017
yearly
20M
10M
0
−10M
−20M
January 1
March 1
May 1
July 1
September 1
N





Case Study 2: Natural Language Processing

In this case study, we will be looking at a dataset of natural-text reviews of wine. We will be using a toolkit called spaCy. spaCy is a Python library for Natural Language Processing (NLP) that is designed to be fast and production-ready. spaCy is a very powerful toolkit, and we will only be scratching the surface of what it can do today.

Load the dataset in the cell below. We will be using the `description` column, which contains the text of the review, and the `points` column, which contains the score given to the wine by the reviewer.

In [15]:
```
df = pd.read_csv("https://github.com/alexwolson/carte_workshop_datasets/raw/main/winemag-data-130k-v2.csv.zip", compressio
```

In [16]:
```
df.head()
```

Out[16]:

	country	description	designation	points	price	province	region_1	region_2	taster_name	taster_twitter_handl
69947	Portugal	Rough around the edges, this is a wine with so...	NaN	86	NaN	Alentejano	NaN	NaN	Roger Voss	@vossroge
100664	Argentina	Attractive, somewhat unusual floral/violet aro...	Reserva	90	17.0	Mendoza Province	Luján de Cuyo	NaN	Michael Schachner	@wineschacl
18773	France	This deliciously warm and ripe wine has swathe...	Tradition	88	13.0	Beaujolais	Beaujolais-Villages	NaN	Roger Voss	@vossroge
76716	Germany	While unabashedly sweet and simple, this lusci...	Dr. L	89	12.0	Mosel	NaN	NaN	Anna Lee C. Iijima	Nal
30554	US	The Stillwater Creek vineyard bottling of Nove...	Stillwater Creek Vineyard	88	28.0	Washington	Columbia Valley (WA)	Columbia Valley	Paul Gregutt	@paulgwin

In [17]:
```
!pip install -U -q "spacy<3.7.0,>=3.6.0"
```

With spaCy, we can use a number of different language models made available for 73 different languages. To make sure that our code runs quickly, we will download the smallest English model, `en_core_web_sm`.

In [18]:
```
!python -m spacy download en_core_web_sm -q
```

```
✔ Download and installation successful
You can now load the package via spacy.load('en_core_web_sm')
```

In [19]:
```
import spacy
from tqdm import tqdm

nlp = spacy.load("en_core_web_sm")
```

The first step in any NLP task is to tokenize the text. Tokenization is the process of breaking up a string into a list of words. When we looked at encoding on Tuesday, the HuggingFace library handled this for us, but spaCy leave us to decide how we want to accomplish this. spaCy provides a `tokenizer` object that we can use to tokenize a string. We can then iterate over the tokens to get the individual words. spaCy also provides a `lemmatizer` object that we can use to get the root form of each word. This is useful because it allows us to group together words that have the same meaning, but different forms (e.g. "run", "runs", "running").

In [20]:
```
tokens = []
lemmas = []
first_doc = nlp(df["description"].iloc[0].lower())
print(f'word       root       part       stop')
print(f'---------------------------------------')
for token in first_doc:
    tokens.append(token.text)
    if not token.is_stop and not token.is_punct:
        lemmas.append(token.lemma_)
    if token.text != token.lemma_:
      print(f'{token.text:10} {token.lemma_:10} {token.pos_:10} {token.is_stop if token.is_stop else ""}')
    else:
      print(f'{token.text:10}            {token.pos_:10} {token.is_stop if token.is_stop else ""}')
```

```
word       root       part       stop
---------------------------------------
rough                 ADJ
around                ADP        True
the                   DET        True
edges      edge       NOUN
,                     PUNCT
this                  PRON       True
is         be         AUX        True
a                     DET        True
wine                  NOUN
with                  ADP        True
some                  DET        True
barnyard              NOUN
flavors    flavor     NOUN
as                    ADV        True
well                  ADV        True
as                    ADP        True
red                   PROPN
berry                 PROPN
fruits     fruit      NOUN
.                     PUNCT
it                    PRON       True
is         be         AUX        True
soft                  ADJ
,                     PUNCT
the                   DET        True
tannins    tannin     NOUN
surrounded surround   VERB
by                    ADP        True
juiciness             NOUN
.                     PUNCT
```

We will apply the process to the entire dataset, using the `nlp.pipe` method. This method allows us to efficiently process a large number of documents. We will also remove stop words, which are words that are very common and don't add much meaning to the text (e.g. "the", "and", "a").

To speed up the process, we will disable the `parser` and `ner` components of the spaCy pipeline. The `parser` component is used to determine the syntactic structure of the text, and the `ner` component is used to identify named entities (e.g. people, places, organizations). Since we are only interested in the tokens, we can disable these components to speed up the process. We are also using the smallest spaCy model, which is faster but less accurate than the larger models. In a production setting, we would likely use a larger model, and a GPU to speed up the process.

In [21]:
```python
tokens = []
for doc in tqdm(nlp.pipe(df["description"].str.lower(), disable=["parser", "ner"]), total=len(df)):
    tokens.append(" ".join([token.lemma_ for token in doc if not token.is_stop and not token.is_punct]))
```

```
100%|████████████████████████████████████████████████████████████████████████████████████████████████████| 64986/
64986 [01:41<00:00, 640.92it/s]
```

Now that we have tokenized the text, we can use it to build a predictive model. We will use the `tokens` column as our input, and the `points` column as our output. We will use a `CountVectorizer` to convert the tokens into a vector of counts. We will then use a `LinearRegression` model to predict the score given to the wine by the reviewer.

In [22]:
```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

model = Pipeline([
    ("vectorizer", CountVectorizer(min_df=0.01)), # Only include words that appear in at least 1% of reviews
    ("regressor", LinearRegression())
])

x_train, x_test, y_train, y_test = train_test_split(tokens, df["points"], test_size=0.2, random_state=42)

start_time = time()
model.fit(x_train, y_train)
print(f'Training time: {time() - start_time} seconds')
```

```
Training time: 0.6703031063079834 seconds
```

In [23]:
```python
print(f'MAE: {mean_absolute_error(y_test, model.predict(x_test)):.2f}')
```

```
MAE: 1.65
```

This is a very strong result! We are able to predict the score given to a wine by the reviewer with a mean absolute error of 1.63 points out of 100. Let's look at the words that are most associated with high and low scores. We can do this by looking at the coefficients of the `LinearRegression` model.

In [24]:
```python
# Get words most associated with high scores
words = model.named_steps["vectorizer"].get_feature_names_out()
coefficients = model.named_steps["regressor"].coef_
word_scores = pd.DataFrame({"word": words, "score": coefficients})
word_scores.sort_values("score", ascending=False).head(10)
```

Out[24]:

	word	score
37	beautiful	1.840969
179	impressive	1.498721
38	beautifully	1.479618
88	complex	1.298041
69	cellar	1.190052
104	delicious	1.148461
274	powerful	1.083830
7	2020	1.082910
120	elegance	1.064420
228	minerality	1.050376

In [25]:
```python
# Get words most associated with low scores
word_scores.sort_values("score", ascending=True).head(10)
```

Out[25]:

	word	score
190	lack	-1.529542
316	simple	-1.252141
304	rustic	-1.073003
313	short	-0.926070
42	bitter	-0.810979
331	sour	-0.682886
170	heavy	-0.634086
28	astringent	-0.627129
321	smell	-0.558750
337	straightforward	-0.552523

In [26]:
```python
# Get top 10 reviews with worst predictions in the test set
test_df = pd.DataFrame({"text": x_test, "actual": y_test, "predicted": model.predict(x_test)})
test_df["error"] = abs(test_df["actual"] - test_df["predicted"])
test_df.sort_values("error", ascending=False)
```

Out[26]:

	text	actual	predicted	error
48898	powerhouse wine drive forward immense tannin s...	98	87.238787	10.761213
227	deep ruby color belie wine cool climate origin...	85	95.583589	10.583589
116141	juicy seductively smooth blockbuster beauty re...	99	90.311588	8.688412
39288	palate open slowly offer initial citrus charac...	98	89.937042	8.062958
111267	ripeness mark pinot flood mouth jammy raspberr...	93	84.938329	8.061671
...	...	...	...	...
51843	white hawk vineyard near los alamos wine nose ...	89	89.001786	0.001786
114899	young perfume wine attractive red fruit wood s...	88	88.001278	0.001278
96551	wine show firm tannin fruit wood acidity ready...	87	86.999132	0.000868
63438	single vineyard wine bodied crisp savory note ...	89	88.999159	0.000841
23522	opening blast juniper eucalyptus pryazinic pal...	86	86.000433	0.000433

12998 rows × 4 columns

Your Turn

Our model predicts the score given to a wine based on the text of the review. But there are a few different columns that we could alternatively predict! Choose one of the following columns, and build a model to predict it based on the text of the review. Explore the results. Do you find anything interesting?

- `country`
- `price`
- `variety`
- `winery`

In [27]:
```python
# Price

model = Pipeline([
    ("vectorizer", CountVectorizer(min_df=0.01)), # Only include words that appear in at least 1% of reviews
    ("regressor", LinearRegression())
])

mean_price = df["price"].dropna().mean()

x_train, x_test, y_train, y_test = train_test_split(tokens, df["price"].fillna(mean_price), test_size=0.2, random_state=42

start_time = time()
model.fit(x_train, y_train)
print(f'Training time: {time() - start_time} seconds')

print(f'MAE: {mean_absolute_error(y_test, model.predict(x_test)):.2f}')

# Get words most associated with high price
words = model.named_steps["vectorizer"].get_feature_names_out()
coefficients = model.named_steps["regressor"].coef_
word_scores = pd.DataFrame({"word": words, "score": coefficients})
display(word_scores.sort_values("score", ascending=False).head(10))

# Get words most associated with low price
display(word_scores.sort_values("score", ascending=True).head(10))
```

```
Training time: 0.6519291400909424 seconds
MAE: 16.33
```

	word	score
382	vintage	14.020459
179	impressive	12.795324
274	powerful	12.099899
69	cellar	11.042178
7	2020	9.313009
90	concentrate	8.959163
37	beautiful	8.756072
1	100	8.655650
273	power	8.441264
378	verdot	8.168089

	word	score
276	price	-10.957232
232	month	-5.733927
3	2016	-5.403677
4	2017	-5.115300
337	straightforward	-4.916342
222	medium	-4.842438
330	soon	-4.435615
152	fruitiness	-3.866246
306	sangiovese	-3.754718
214	low	-3.741048

Case Study 3: Recommendation

For our last case study, we are going to look at a dataset of movie ratings. We're going to start by building a simple recommendation system that recommends movies by finding the most similar users. Then, we will move on to using a powerful library that implements some of the state-of-the-art approaches.

Let's begin by loading part of the MovieLens dataset. This is a popular dataset of user ratings of movies. We will be using the `ratings` dataset, which contains the ratings given by users to movies. We will also load the `movies` dataset, which contains information about each movie.

In [28]:
```
movies = pd.read_csv("https://github.com/alexwolson/carte_workshop_datasets/raw/main/movies.csv.zip", compression="zip")
ratings = pd.read_csv("https://github.com/alexwolson/carte_workshop_datasets/raw/main/ratings.csv.zip", compression="zip")
```

In [29]:
```
movies.head()
```

Out[29]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

In [30]:
```
ratings.head()
```

Out[30]:

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931

As you can see, the `ratings` dataset contains a `userId`, a `movieId`, a `rating`, and a `timestamp`. The `movies` dataset contains a `movieId`, a `title`, and a list of `genres`. We are not going to make predictions based on genre today, but it's a common approach to recommendation in this area. Instead, we will just focus on the users and their ratings.

Let's look at a random user to get a sense of what a users' ratings could look like:

In [31]:
```
ratings[ratings.userId == 42].merge(movies, on="movieId") # Merging so that we can see what the movies are
```

Out[31]:

	userId	movieId	rating	timestamp	title	genres
0	42	3	4.0	996221045	Grumpier Old Men (1995)	Comedy|Romance
1	42	7	3.0	996220162	Sabrina (1995)	Comedy|Romance
2	42	10	5.0	996215205	GoldenEye (1995)	Action|Adventure|Thriller
3	42	11	5.0	996219314	American President, The (1995)	Comedy|Drama|Romance
4	42	16	5.0	996218017	Casino (1995)	Crime|Drama
...	...	...	...	...	...	...
435	42	4623	4.0	996258272	Major League (1989)	Comedy
436	42	4629	2.0	996260295	Next of Kin (1989)	Action|Crime|Thriller
437	42	4654	3.0	996260295	Road House (1989)	Action|Drama
438	42	4679	4.0	996258824	Uncle Buck (1989)	Comedy
439	42	4686	2.0	996256937	Weekend at Bernie's II (1993)	Adventure|Comedy

440 rows × 6 columns

If we are working with users who have already rated a number of movies on the system, one approach is to look for the most similar users, and then recommend movies that those users have rated highly. We can do this by computing the similarity between users. We will use the cosine similarity between the ratings of two users as our measure of similarity. The cosine similarity is a measure of the angle between two vectors. If the angle is small, the vectors are similar. If the angle is large, the vectors are dissimilar. We will use the `cosine_similarity` function from the `sklearn.metrics.pairwise` module to compute the cosine similarity between users.

We will also need to convert the format of our data from a list of users and reviews, to a matrix of users and reviews. We can do this using the `pivot_table` method. This method takes a DataFrame, and converts it from a long format to a wide format. We will use the `userId` as the index, the `movieId` as the columns, and the `rating` as the values. We will also fill in any missing values with 0, since we are only interested in whether a user has rated a movie or not.

In [32]:
```
from sklearn.metrics.pairwise import cosine_similarity

ratings_matrix = ratings.pivot_table(index="userId", columns="movieId", values="rating", fill_value=0)
```

In [33]:
```python
user_one = ratings_matrix.iloc[42]
user_two = ratings_matrix.iloc[43]
print(f'Cosine similarity between user 42 and user 43: {cosine_similarity([user_one], [user_two])[0][0]:.2f}')
```

```
Cosine similarity between user 42 and user 43: 0.06
```

Now that we have our matrix and our method, let's go ahead and compute the similarity between each pair of users. We will store the results in a DataFrame, with the `userId` as the index and the `similarity` as the value.

In [34]:
```python
from sklearn.metrics.pairwise import cosine_similarity
from scipy.sparse import csr_matrix
import numpy as np

# Convert the ratings matrix to a sparse matrix format if not already
ratings_sparse = csr_matrix(ratings_matrix.values)

# Compute the cosine similarity matrix in a vectorized way
# This computes the full n x n similarity matrix
similarities = cosine_similarity(ratings_sparse)

# Since the similarity with itself is always 1, we can fill the diagonal with 1s
np.fill_diagonal(similarities, 1)
```

Now that we have the similarity between each pair of users, we can use it to make recommendations. For user 42, we can take the top 10 users who are most similar, and then recommend the movies that they have rated most highly.

In [35]:
```python
similarities_df = pd.DataFrame(similarities, index=ratings_matrix.index, columns=ratings_matrix.index)
```

In [36]:
```python
similar_users = similarities_df[42].sort_values(ascending=False).head(10)
```

In [37]:
```python
recommended_movies = ratings_matrix.loc[similar_users.index].mean().sort_values(ascending=False)
# Remove movies that the user has already rated
recommended_movies = recommended_movies[~recommended_movies.index.isin(ratings_matrix.iloc[42].replace(0, np.nan).dropna()
```

In [38]:
```python
for movie_id, rating in recommended_movies.head(10).items():
    print(f'{movies[movies["movieId"] == movie_id]["title"].iloc[0]} ({rating:.2f})')
```

```
Matrix, The (1999) (4.75)
Saving Private Ryan (1998) (4.55)
Star Wars: Episode IV - A New Hope (1977) (4.40)
Silence of the Lambs, The (1991) (4.35)
Star Wars: Episode VI - Return of the Jedi (1983) (4.35)
Godfather, The (1972) (4.20)
Star Wars: Episode V - The Empire Strikes Back (1980) (4.15)
Goodfellas (1990) (4.10)
American Beauty (1999) (4.05)
Princess Bride, The (1987) (3.95)
```

And there we have it - a simple recommendation system! Unfortunately, this approach has some major problems.

1. Scalability - while it doesn't take too long to calculate similarities between 600 or so users, company like Netflix has millions or even billions of users!
2. Cold start - what if we have a new user who hasn't rated any movies yet? We can't make any recommendations for them.
3. Popularity bias - this approach will recommend popular movies, since lots of people have rated them, even if they are not a good fit for the user.

Let's use the same data, but employ a more sophisticated approach. We will use a library called Surprise, which implements a number of state-of-the-art methods for recommendation.

In [39]:
```python
!pip install -U -q surprise
```

First, we have to convert the data into a format that Surprise can understand. We will use the `Reader` class to specify the range of ratings, and then use the `Dataset` class to convert the data.

In [40]:
```python
from surprise import Dataset, Reader, SVD

reader = Reader(rating_scale=(0.5, 5.0))
data = Dataset.load_from_df(ratings[["userId", "movieId", "rating"]], reader)
```

We are going to use Singular Value Decomposition, or SVD. SVD works by breaking down our single, huge user-movie matrix into three smaller matrices. This process allows us to capture the most important patterns in the data using fewer details, which is essential when working with millions or even *billions* of users. Using these three smaller matrices, SVD can approximate the expected values for missing entries in the user-movie matrix. This allows us to make predictions for new users, and to make recommendations for movies that have not been rated by many users.

In [41]:
```python
model = SVD(random_state=42)
start_time = time()
```

```
model.fit(data.build_full_trainset())
print(f'Training time: {time() - start_time} seconds')
```

```
Training time: 0.5968313217163086 seconds
```

In [42]:
```
# Get top 10 movies for user 42
user_42_movies = ratings[ratings["userId"] == 42]["movieId"].unique()
predicted_ratings = []
for movie_id in movies["movieId"].unique():
    if movie_id in user_42_movies:
        continue
    predicted_ratings.append((movie_id, model.predict(42, movie_id).est))
predicted_ratings.sort(key=lambda x: x[1], reverse=True)
```

In [43]:
```
for movie_id, rating in predicted_ratings[:10]:
    print(f'{movies[movies["movieId"] == movie_id]["title"].iloc[0]} ({rating:.2f})')
```

```
Patton (1970) (4.78)
Life Is Beautiful (La Vita è bella) (1997) (4.75)
Inception (2010) (4.74)
Dark Knight, The (2008) (4.68)
3:10 to Yuma (2007) (4.63)
Monty Python and the Holy Grail (1975) (4.59)
Lawrence of Arabia (1962) (4.58)
Boondock Saints, The (2000) (4.57)
Mary Poppins (1964) (4.57)
Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981) (4.56)
```

As you can see, while many of these films are certainly popular, the SVD approach allows us to recommend movies that are more tailored to the user. We can also use the model to predict the rating that a user will give to a movie. This is a good way of evaluating the quality of the model.

In [44]:
```
ratings[ratings["userId"] == 42].merge(movies, on="movieId") # Merging so that we can see what the movies are
```

Out[44]:

	userId	movieId	rating	timestamp	title	genres
0	42	3	4.0	996221045	Grumpier Old Men (1995)	Comedy|Romance
1	42	7	3.0	996220162	Sabrina (1995)	Comedy|Romance
2	42	10	5.0	996215205	GoldenEye (1995)	Action|Adventure|Thriller
3	42	11	5.0	996219314	American President, The (1995)	Comedy|Drama|Romance
4	42	16	5.0	996218017	Casino (1995)	Crime|Drama
...	...	...	...	...	...	...
435	42	4623	4.0	996258272	Major League (1989)	Comedy
436	42	4629	2.0	996260295	Next of Kin (1989)	Action|Crime|Thriller
437	42	4654	3.0	996260295	Road House (1989)	Action|Drama
438	42	4679	4.0	996258824	Uncle Buck (1989)	Comedy
439	42	4686	2.0	996256937	Weekend at Bernie's II (1993)	Adventure|Comedy

440 rows × 6 columns

In [45]:
```
predictions = []
for movie_id in user_42_movies:
    predictions.append({
        "movieId": movies[movies["movieId"] == movie_id]["title"].iloc[0],
        "predicted": model.predict(42, movie_id).est,
        "actual": ratings[(ratings["userId"] == 42) & (ratings["movieId"] == movie_id)]["rating"].iloc[0]
    })
predictions_df = pd.DataFrame(predictions)
predictions_df["error"] = abs(predictions_df["predicted"] - predictions_df["actual"])
print(f'MAE: {predictions_df["error"].mean():.2f}')
```

```
MAE: 0.58
```

Let's compare this against our original method:

In [46]:
```
similar_users = similarities_df[42].sort_values(ascending=False).head(10)
recommended_movies = ratings_matrix.loc[similar_users.index].mean().sort_values(ascending=False)

predictions = []
for movie_id in user_42_movies:
    predictions.append({
        "movieId": movies[movies["movieId"] == movie_id]["title"].iloc[0],
        "predicted": recommended_movies[movie_id],
        "actual": ratings[(ratings["userId"] == 42) & (ratings["movieId"] == movie_id)]["rating"].iloc[0]
    })
predictions_df = pd.DataFrame(predictions)
```

```python
predictions_df["error"] = abs(predictions_df["predicted"] - predictions_df["actual"])
print(f'MAE: {predictions_df["error"].mean():.2f}')
```

```
MAE: 1.93
```

As we can see, the SVD approach is not only much faster, but more accurate in reproducing the user's original ratings. SVD is simple, effective, and highly scalable - which is why it was the industry standard for companies like Amazon, Netflix, and Spotify for many years.

Your Turn

One challenge with a five-star rating system (and a big reason why companies like YouTube and Netflix have long since moved to a 'thumbs up, thumbs down' approach) is that each user has a different idea of what each rating means. For example, one user might give a 5-star rating to their favourite movie, while another user might only give a 5-star rating to a movie that they consider to be perfect. Try setting all ratings to 1 if a user rated 4 or 5, or 0 otherwise. How does this affect prediction quality?

In [47]:
```python
# Replace ratings with 1 if 4 or 5, 0 otherwise
ratings["rating"] = ratings["rating"].apply(lambda x: 1 if x >= 4 else 0)

reader = Reader(rating_scale=(0, 1))
data = Dataset.load_from_df(ratings[["userId", "movieId", "rating"]], reader)

model = SVD(random_state=42)

start_time = time()
model.fit(data.build_full_trainset())
print(f'Training time: {time() - start_time} seconds')

# Get top 10 movies for user 42
user_42_movies = ratings[ratings["userId"] == 42]["movieId"].unique()

predicted_ratings = []
for movie_id in movies["movieId"].unique():
    if movie_id in user_42_movies:
        continue
    predicted_ratings.append((movie_id, model.predict(42, movie_id).est))

predicted_ratings.sort(key=lambda x: x[1], reverse=True)

for movie_id, rating in predicted_ratings[:10]:
    print(f'{movies[movies["movieId"] == movie_id]["title"].iloc[0]} ({rating:.2f})')

# Get accuracy

predictions = []
for movie_id in user_42_movies:
    predictions.append({
        "movieId": movies[movies["movieId"] == movie_id]["title"].iloc[0],
        "predicted": model.predict(42, movie_id).est,
        "actual": ratings[(ratings["userId"] == 42) & (ratings["movieId"] == movie_id)]["rating"].iloc[0]
    })

predictions_df = pd.DataFrame(predictions)
predictions_df["error"] = abs(predictions_df["predicted"] - predictions_df["actual"])
print(f'MAE: {predictions_df["error"].mean():.2f}')
```

```
Training time: 0.6137218475341797 seconds
My Fair Lady (1964) (1.00)
Mary Poppins (1964) (1.00)
Dial M for Murder (1954) (1.00)
Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981) (1.00)
12 Angry Men (1957) (1.00)
Grand Day Out with Wallace and Gromit, A (1989) (1.00)
This Is Spinal Tap (1984) (1.00)
Rosemary's Baby (1968) (1.00)
Monty Python's And Now for Something Completely Different (1971) (1.00)
Gilda (1946) (1.00)
MAE: 0.38
```

Conclusion, and bonus

We have covered a lot of ground today! We have looked at three different case studies, each of which uses a different approach to AI in Market Strategy. We have seen how AI can be used to predict the future, to understand text, and to make recommendations. We've looked at not just real-world examples, but also state-of-the-art toolkits that are used by many companies today.

As a bonus exercise, pick the case study that you found most interesting, and see if you can expand on the results from today. Some ideas:

- Predictive Analytics: Can you visualize the data in a more informative way? Can you predict the volume of avocados sold for a specific region, or for a specific type of avocado?

- Natural Language Processing: Can you identify reviewers' favourite regions or varieties of wine? Can you identify the most common words used to describe different types of wine?
- Recommendation: Surprise includes a number of different models. Can you try a different model, and compare the results? Can you use the model to recommend movies to a new user?