Predicting delivery times using regression

1 Introduction

Wolt is a food delivery company that provides a platform where people can order food from restaurants. When an order is placed on the app, Wolt dispatches a driver to pick up the order and deliver it to the user. Recently, the company shared some data on orders placed on its app in the downtown Helsinki area in August and September of 2020. As someone who uses their app, I was curious to find out how good Wolt is at predicting delivery times – my experience (a small sample, admittedly) is that Wolt regularly fails to be accurate in its predictions. I decided to attempt predicting delivery minutes for orders and compare the results to Wolt's predictions.

The report proceeds as follows. In section 2, I discuss the problem and data. Section 3 introduces the methods used. In section 4, I review the results of using these methods and compare them to Wolt's estimation errors. Section 5 presents the conclusions, and is followed by some additional visualizations and the code in the appendix.

2 Problem formulation and data

The data points are food delivery orders placed using the Wolt app. There were 18,706 observations in the dataset and 13 variables to describe each order. These variables are listed in table 4 in the appendix. **The goal of this project is to predict delivery times using regression methods. The label is delivery minutes. The features are distance, number of items in the order, hour of the day, and day of the week.** The resulting prediction errors will be compared to Wolt's, which is a natural benchmark for my models.

Table 1: Features used to predict delivery minutes.

Variable name	Type
Distance	Float
Items	Integer
Hour	Integer
Monday	Binary
Tuesday	Binary
Wednesday	Binary
Thursday	Binary
Friday	Binary
Saturday	Binary
Sunday	Binary

The relationships between the original variables (which are listed in table 4 in the appendix) were weak aside from obvious associations between coordinates (see figure 1). Of the original variables, I only used the number of items, since more items in an order could mean longer preparation time for the venue. Then I resorted to feature engineering. I figured that the distance between user

and venue would be a significant factor of delivery times. I approximated this distance using the haversine formula for calculating distances on the surface of a sphere. The haversine distance can be described as the 'as the crow flies' distance. Wolt uses it to price its deliveries.

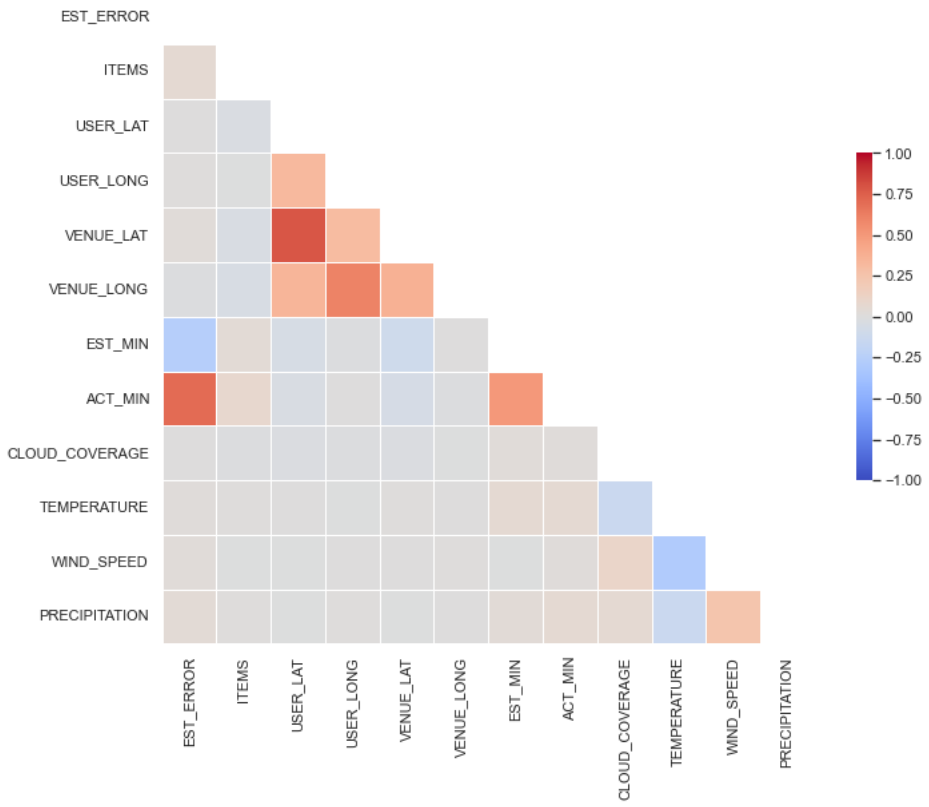


Figure 1: Correlations of the original variables.

From the timestamps I extracted the hour of day, ranging from 1 to 24, and the day of the week, ranging from Monday to Sunday, to use as features. Demand for deliveries varies with the time of day and day of the week, and my reasoning was that higher demand could lead to longer delivery times if it's not met with a higher supply of drivers. The heatmap in figure 5 indicates that the peak demand hours are in the afternoon. Demand in the morning depends more on the day of the week.

3 Methods

I used NumPy arrays to represent the features and the label. 60 percent of data was used for training, 20 percent for validation, and 20 percent for testing. I had enough data to manage without 10-fold cross validation, and the results were the same either way. As my methods, I chose Lasso and decision tree regression. Lasso performs regularization as well as feature selection, both of which seem useful properties in this case. I have not used tree regression before, and the novelty of it made me want to try it.

3.1 Lasso regression

The hypothesis space in Lasso regression is

$$H^{(10)} := \{h^{(w)} : \mathbb{R}^{10} \to \mathbb{R} : h^{(w)} := w^T x \text{ with some parameter vector } w \in \mathbb{R}^{10}\},$$

and the loss function L to be minimized is given by

$$L\left((x, y), h^{(w)}\right) = \frac{1}{m}\sum_{i=1}^{m}(y - w^T x)^2 + \lambda||w||_1,$$

where we have the penalty term $\lambda||w||_1$.

3.2 Decision tree regression

Hypothesis space is formed by all possible decision trees this data could generate. I used absolute error loss, hoping it would make the model more robust. This seemed sensible in light of all the warnings about decision trees tending to overfit. I tried maximum tree depths between 1 and 13.

4 Results

4.1 Lasso

Training and testing errors for different values of the scaling factor λ are visualized in figure 2. The lowest validation error was achieved with $\lambda = 0.001$, so that is the value I used for testing later.

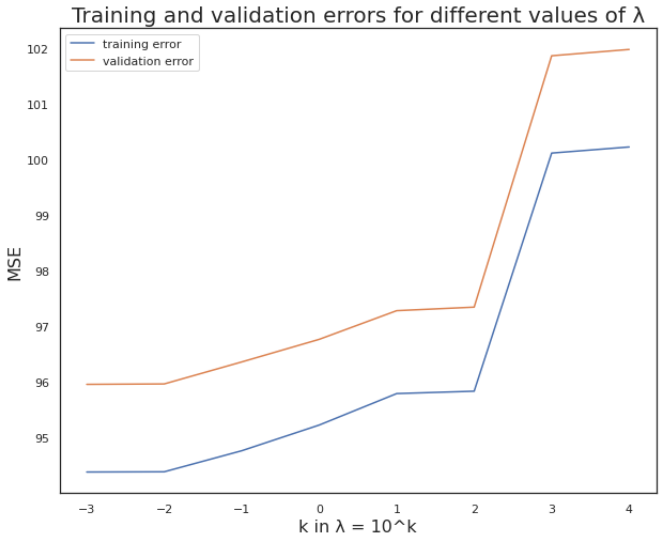


Figure 2: Training and validation errors for different values of scaling factor λ in the Lasso model.

Lasso's ability to do feature selection is illustrated in table 2. As the value of the scaling factor λ is increased, more and more coefficients take the value zero and the intercept starts to approach

the mean actual delivery minutes in the data. For larger values of λ, Lasso deems the variables corresponding to zero coefficients as irrelevant to predicting the label value. Distance and the number of items remain as relevant features for higher values of λ, until they too are assigned zero coefficients. For $\lambda = 10^4$, only the intercept remains.

Table 2: Lasso intercept and coefficients for different values of scaling factor λ.

λ	Intercept	Distance	Items	Hour	Mon	Tue	Wed	Thu	Fri	Sat	Sun
10^{-3}	27.79	0.004	0.480	0.046	1.214	0	-0.282	-0.326	1.057	-0.660	0.985
10^{-2}	27.72	0.004	0.478	0.044	1.137	0	-0.216	-0.250	0.994	-0.607	0.936
10^{-1}	28.18	0.004	0.458	0.038	0.059	0	0	0	0.058	-0.394	0.135
10^{0}	29.36	0.004	0.207	0	0	0	0	0	0	0	0
10^{1}	29.95	0.004	0	0	0	0	0	0	0	0	0
10^{2}	30.18	0.004	0	0	0	0	0	0	0	0	0
10^{3}	32.52	5e-5	0	0	0	0	0	0	0	0	0
10^{4}	32.55	0	0	0	0	0	0	0	0	0	0

Table 2 also highlights an error I have made by ignoring the assumption about linearity. Originally I one-hot encoded Hour, splitting the day into three blocks (morning, afternoon, and evening), but abandoned the idea because I wanted fewer features. For smaller values of λ, Hour has a small positive coefficient. It assumes the relationship between the hour and delivery minutes is linear, but it doesn't really make sense to assume that deliveries at hour = 24 necessarily take more time than at hour = 1, or that the difference in minutes is 23 times the coefficient for Hour, all other things being equal. Also, it's interesting to note that the coefficients for Wednesday, Thursday, and Saturday are negative, meaning they have a negative effect on delivery minutes, whereas for the other days they are positive. As I counted daily order counts for the heatmap in figure 5, it looked like these days of the week tended to have higher demand – so perhaps the demand-supply equilibrium is indeed somehow reflected in delivery minutes.

Figure 3: Training and validation errors for different values of maximum tree depth.

4.2 Decision tree regression

I tried decision tree regression with different maximum tree depths. The resulting training and validation errors are presented in figure 3. The lowest validation error was reached with maximum tree depth of 6, so I chose that as the decision tree model to use in model comparison.

4.3 Model comparison

Finally, I compare the best versions of my models – Lasso with scaling factor $\lambda = 0.001$ and tree regression with maximum depth of 6 – and Wolt's estimates. As the original dataset included actual delivery minutes as well as Wolt's estimated delivery minutes, I actually have a benchmark to compare my results to. The mean squared error (MSE) for Wolt's predictions was

$$E_{\text{ref}} = \text{MSE}_{\text{Wolt}} \approx 82$$

What matters to customers is not that their orders are *exactly on time*, but rather that they are *approximately on time*, which is why in table 3 I also compare the shares of orders for which estimated delivery minutes were (1) equal to actual delivery minutes, (2) within 5 minutes of actual delivery minutes, and (3) within 10 minutes of actual delivery minutes.

Table 3: Model error comparison.

Model	MSE	% exactly on time	% within 5 min	% within 10 min
Wolt	82.0	4.2	45.1	75.1
Lasso	93.6	0.0	35.8	66.9
Decision tree	92.3	3.9	41.2	70.4

MSE was 93.6 for Lasso and 92.3 for tree regression. Both are higher than the benchmark 82. Decision tree regression is not that much worse than Wolt when looking at what share of orders would have had estimated delivery times at most 5 or 10 minutes off the actual delivery times. This, however, is probably no great victory and rather just due to most orders being pretty average – it's the occasional orders that are very late or early that are difficult to predict. The conclusion is that my models probably underfit, or at the very least don't perform as well as Wolt's.

5 Conclusions

Tree regression outperformed Lasso by a hair's breadth. Of the ten features used to predict the label, distance and number of items were most important. Interpreting the relevance of the day of the week and hour of the day was more challenging. Both models had pretty poor performance, which I anticipated, but decided not to agonize over – I would wager that improving predictions is a task with diminishing returns. Wolt's business is built on streamlining operations, and they must have spent considerable efforts to chip away at their model error. It pays for them to obsess over their MSE scores, but I'm content with these results. If I had an incentive to spend more time on this, I could incorporate more features into the models – traffic data could be an interesting addition to the model.

Sources

Sklearn: Decision tree regression

McElreath, Richard. Statistical rethinking: A Bayesian course with examples in R and Stan. Chapman and Hall/CRC, 2020.

Sklearn: Lasso regression

Wikipedia: Haversine formula

Wolt data

Appendix: Visualizations and Code

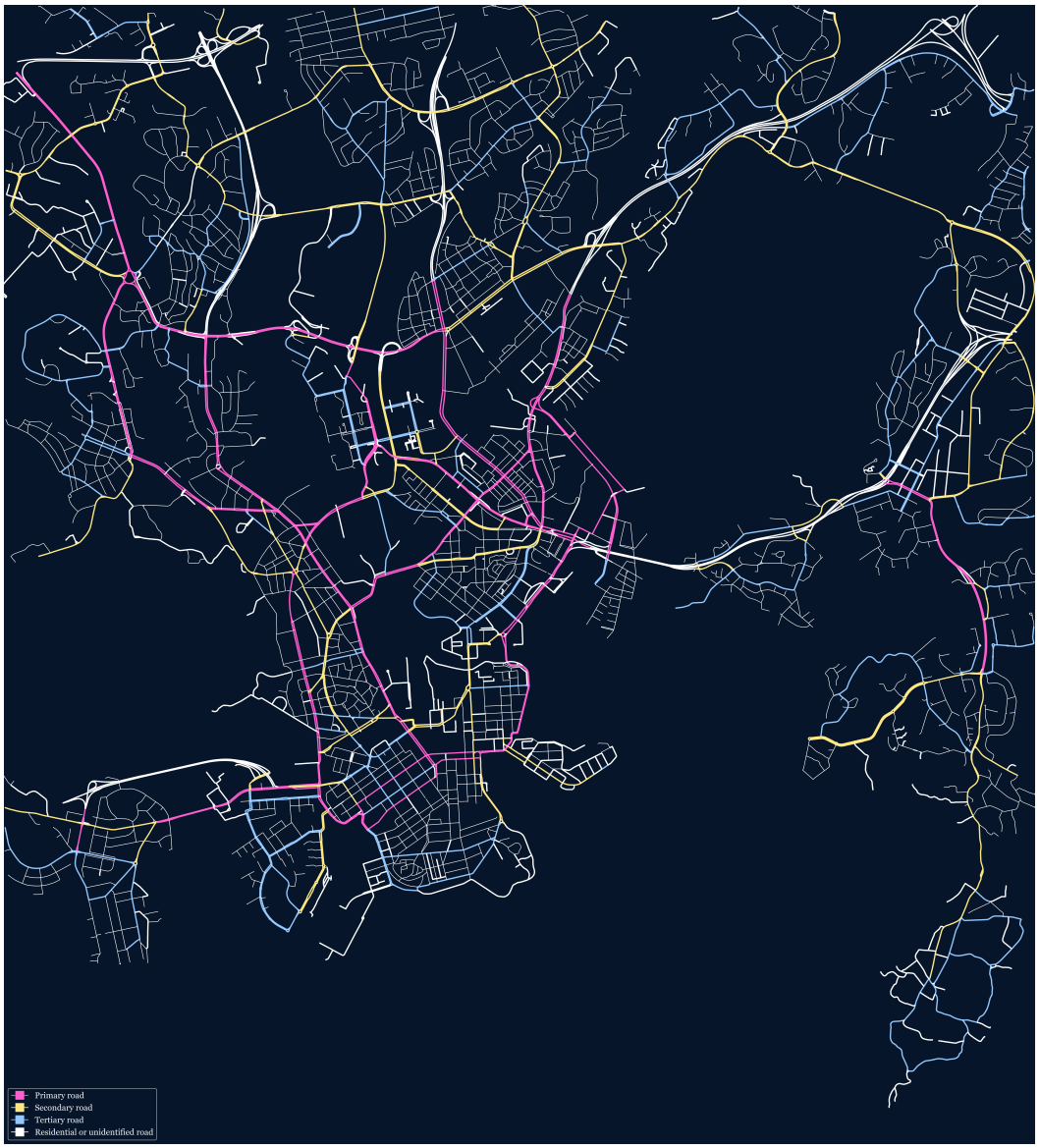


Figure 4: Downtown Helsinki. All users and venues were located within this area.

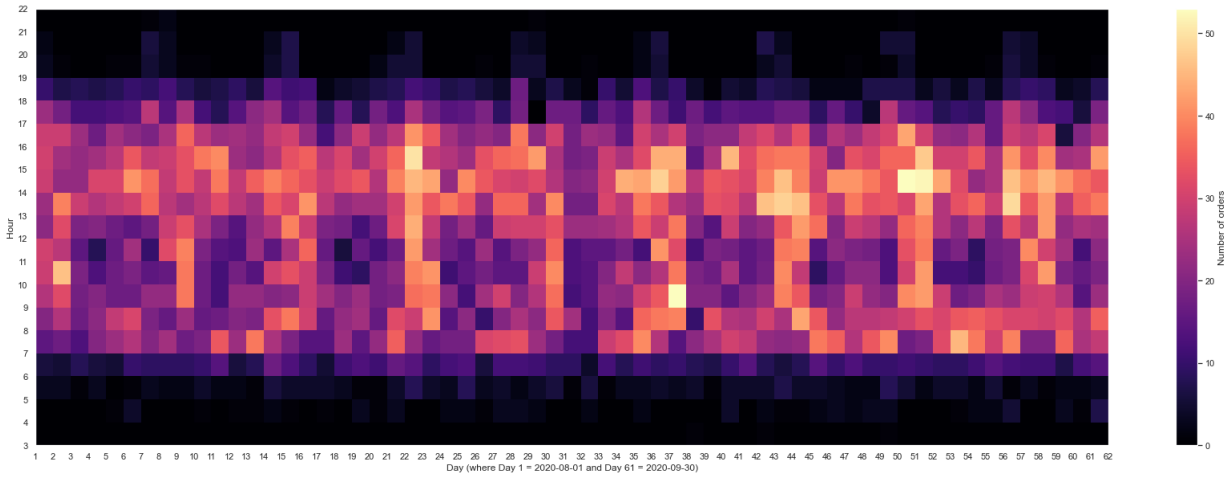


Figure 5: Order counts by the date and the hour. Lighter colors indicate more orders, darker colors fewer orders.

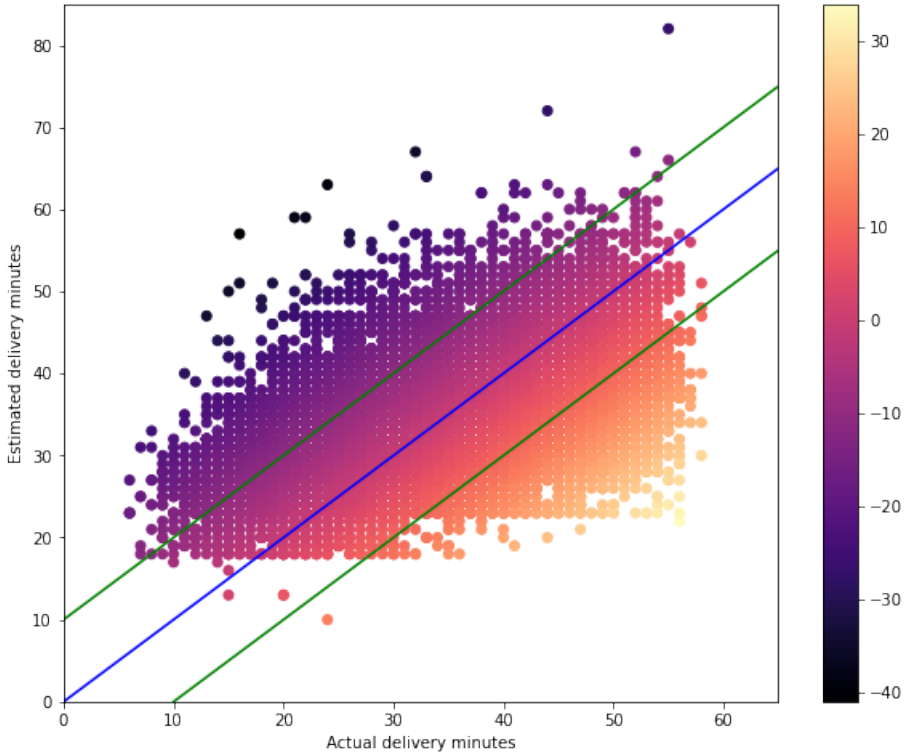


Figure 6: Orders according to how the actual delivery time compared with the estimated delivery time. Blue line corresponds to orders whose ETA was correct. Data points between the two green lines were delivered within 10 minutes of the estimated time.

In figure 2, I plotted actual delivery minutes against Wolt's estimated delivery minutes. The color of the data points indicates how late or early the order was, with yellows indicating lateness and purples earliness. The blue line depicts orders that were exactly on time. Data points in the area between the two green lines are orders which were early or late by at most 10 minutes

of their estimated delivery time. Data points outside the area depict orders for which Wolt's estimated delivery minutes was off by over 10 minutes. We see that there is a lot of scatter in the data, meaning Wolt's predictions are often not that accurate, which echoes my own experience. This should illustrate how challenging it is to make predictions when unexpected issues with the restaurant or driver can delay the delivery.

Table 4: Original variables in the dataset

Variable	Abbreviation	Type
Timestamp	-	Object
Actual delivery minutes	ACT_MIN	Integer
Estimated delivery minutes	EST_MIN	Integer
Actual delivery minutes - estimated delivery minutes	EST_ERROR	Integer
Item count	ITEMS	Integer
User latitude	USER_LAT	Float
User longitude	USER_LONG	Float
Venue latitude	VENUE_LAT	Float
Venue longitude	VENUE_LONG	Float
Cloud coverage	CLOUD_COVERAGE	Float
Temperature	TEMPERATURE	Float
Wind speed	WIND_SPEED	Float
Precipitation	PRECIPITATION	Float

ML course project

March 25, 2022

0.0.1 Import relevant modules

[1]:
```
# export
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import date
import calendar
import matplotlib.cm as cm
import matplotlib.colors as colors
from matplotlib.lines import Line2D
import math
from sklearn.linear_model import LinearRegression, Lasso, Ridge, HuberRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, KFold
from sklearn.utils import shuffle
```

1 Data

[2]:
```
### Load the data
```

[3]:
```
orders2020 = pd.read_csv("orders_autumn_2020.csv")
```

[4]:
```
### Describe the data
```

[5]:
```
orders2020.head()
```

[5]:
```
                  TIMESTAMP  \
0  2020-08-01 06:07:00.000
1  2020-08-01 06:17:00.000
2  2020-08-01 06:54:00.000
3  2020-08-01 07:09:00.000
4  2020-08-01 07:10:00.000

   ACTUAL_DELIVERY_MINUTES - ESTIMATED_DELIVERY_MINUTES  ITEM_COUNT  USER_LAT  \
```

```
0                                                -19              1    60.158
1                                                 -7              8    60.163
2                                                -17              4    60.161
3                                                 -2              3    60.185
4                                                 -1              2    60.182

   USER_LONG  VENUE_LAT  VENUE_LONG  ESTIMATED_DELIVERY_MINUTES  \
0     24.946     60.160      24.946                          29
1     24.927     60.153      24.910                          39
2     24.937     60.162      24.939                          23
3     24.954     60.190      24.911                          28
4     24.955     60.178      24.949                          27

   ACTUAL_DELIVERY_MINUTES  CLOUD_COVERAGE  TEMPERATURE  WIND_SPEED  \
0                       10             0.0         15.0     3.53644
1                       32             0.0         15.0     3.53644
2                        6             0.0         15.0     3.53644
3                       26             0.0         16.7     3.52267
4                       26             0.0         16.7     3.52267

   PRECIPITATION
0            0.0
1            0.0
2            0.0
3            0.0
4            0.0
```

[6]:
```
orders2020.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18706 entries, 0 to 18705
Data columns (total 13 columns):
 #   Column                                          Non-Null Count  Dtype
---  ------                                          --------------  -----
 0   TIMESTAMP                                       18706 non-null
object
 1   ACTUAL_DELIVERY_MINUTES - ESTIMATED_DELIVERY_MINUTES  18706 non-null  int64
 2   ITEM_COUNT                                      18706 non-null  int64
 3   USER_LAT                                        18706 non-null
float64
 4   USER_LONG                                       18706 non-null
float64
 5   VENUE_LAT                                       18706 non-null
float64
 6   VENUE_LONG                                      18706 non-null
float64
 7   ESTIMATED_DELIVERY_MINUTES                      18706 non-null  int64
```

```
 8   ACTUAL_DELIVERY_MINUTES                          18706 non-null  int64
 9   CLOUD_COVERAGE                                   18429 non-null
float64
 10  TEMPERATURE                                      18429 non-null
float64
 11  WIND_SPEED                                       18429 non-null
float64
 12  PRECIPITATION                                    18706 non-null
float64
dtypes: float64(8), int64(4), object(1)
memory usage: 1.9+ MB
```

[7]:
```
orders2020.describe()
```

[7]:

	ACTUAL_DELIVERY_MINUTES - ESTIMATED_DELIVERY_MINUTES	ITEM_COUNT
count	18706.000000	18706.000000
mean	-1.201058	2.688228
std	8.979834	1.886455
min	-41.000000	1.000000
25%	-7.000000	1.000000
50%	-2.000000	2.000000
75%	5.000000	4.000000
max	34.000000	11.000000

	USER_LAT	USER_LONG	VENUE_LAT	VENUE_LONG
count	18706.000000	18706.000000	18706.000000	18706.000000
mean	60.175234	24.941244	60.175643	24.941214
std	0.012674	0.016540	0.011509	0.014482
min	60.153000	24.909000	60.149000	24.878000
25%	60.163000	24.926000	60.167000	24.930000
50%	60.175000	24.943000	60.170000	24.941000
75%	60.186000	24.954000	60.186000	24.950000
max	60.201000	24.980000	60.219000	25.042000

	ESTIMATED_DELIVERY_MINUTES	ACTUAL_DELIVERY_MINUTES	CLOUD_COVERAGE
count	18706.000000	18706.000000	18429.000000
mean	33.809313	32.608254	11.996853
std	7.340283	10.018879	23.812605
min	10.000000	6.000000	0.000000
25%	28.000000	25.000000	0.000000
50%	33.000000	32.000000	0.000000
75%	38.000000	40.000000	25.000000
max	82.000000	58.000000	100.000000

	TEMPERATURE	WIND_SPEED	PRECIPITATION
count	18429.000000	18429.000000	18706.000000
mean	16.973536	3.790991	0.332756

```
std        3.411900      1.456017      1.129234
min        6.100000      0.077419      0.000000
25%       14.400000      2.696190      0.000000
50%       16.700000      3.631970      0.000000
75%       18.900000      4.692530      0.000000
max       26.700000      9.857300      6.315790
```

1.0.1 Modify the data

Adding variables time, date, weekday, and hour of day

[8]:
```
def weekday(row):
    date = row.loc['DATE']
    day = calendar.day_name[date.weekday()]
    return day
```

[9]:
```
def hour_of_day(row):
    hour = int(row.loc['TIME'].strftime("%H"))+1
    return hour
```

[10]:
```
orders = orders2020
orders = orders.rename(columns={'ACTUAL_DELIVERY_MINUTES -␣
↪ESTIMATED_DELIVERY_MINUTES': 'EST_ERROR',
                                'ITEM_COUNT':'ITEMS',␣
↪'ESTIMATED_DELIVERY_MINUTES': 'EST_MIN',
                                'ACTUAL_DELIVERY_MINUTES': 'ACT_MIN'})

orders['DATE'] = pd.to_datetime(orders['TIMESTAMP']).dt.date
orders['TIME'] = pd.to_datetime(orders['TIMESTAMP']).dt.time
orders['WEEKDAY'] = orders.apply(weekday, axis=1)                    #␣
↪We match each date to a weekday
orders['HOUR_OF_DAY'] = orders.apply(hour_of_day,axis=1)
```

Adding distances between users and venues

[11]:
```
def beeline_distance(row):
    """ Returns the distance between a user and a venue using the haversine␣
↪formula on their coordinates. """

    user_lat = row['USER_LAT']
    user_long = row['USER_LONG']
    venue_lat = row['VENUE_LAT']
    venue_long = row['VENUE_LONG']

    R = 6371000                          # Earth's radius in meters
     1 = user_lat * math.pi/180          # Latitude in radians
     2 = venue_lat * math.pi/180         # Latitude in radians
    Δ  = ( 1- 2) * math.pi/180           # Difference in latitudes, radians
```

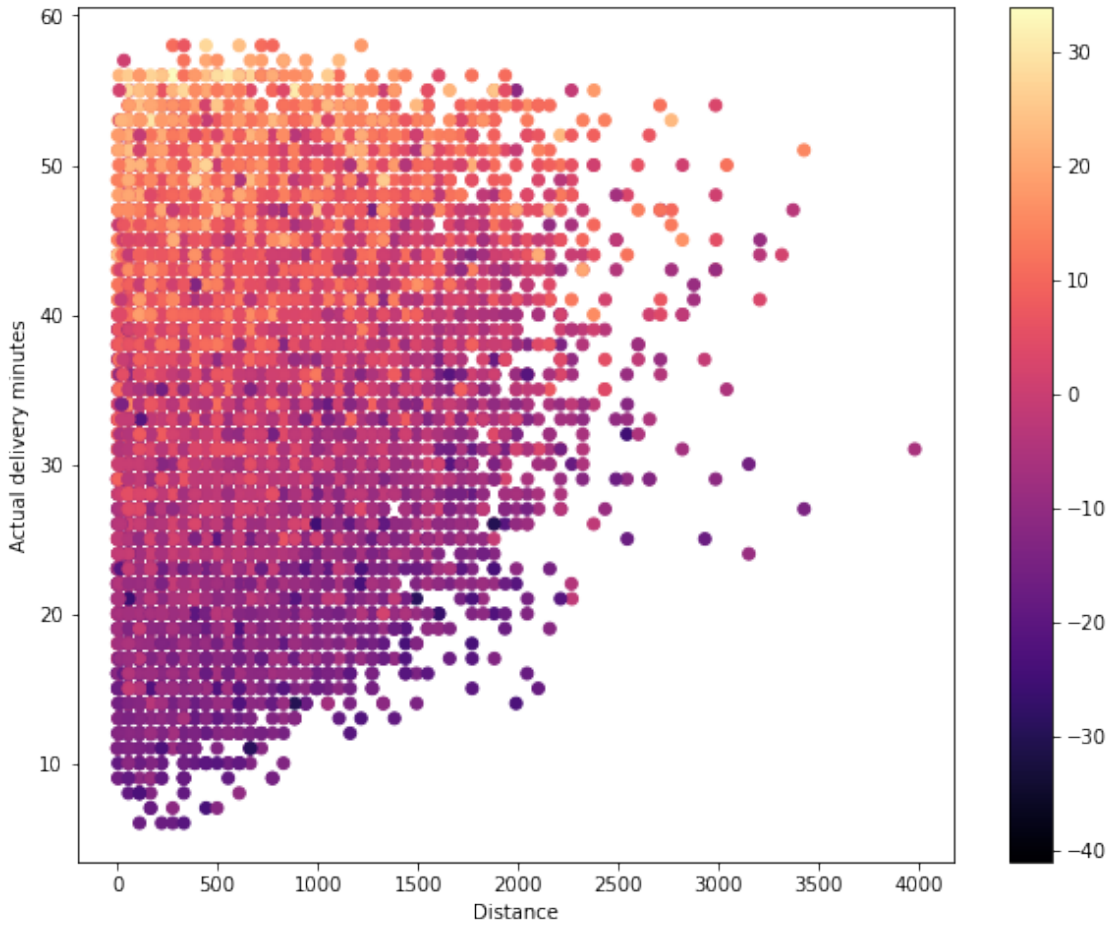
```
    Δ  = (user_long-venue_long) * math.pi/180       # Difference in longitudes,␣
↪radians

    a = math.sin(Δ /2) * math.sin(Δ /2) + math.cos( 1) * math.cos( 2) * math.
↪sin(Δ /2) * math.sin(Δ /2);
    c = 2 * math.atan2(math.sqrt(a),math.sqrt(1-a))
    dist = R*c

    return dist
```

[12]:
```
orders['DISTANCE'] = orders.apply(beeline_distance,axis=1)
```

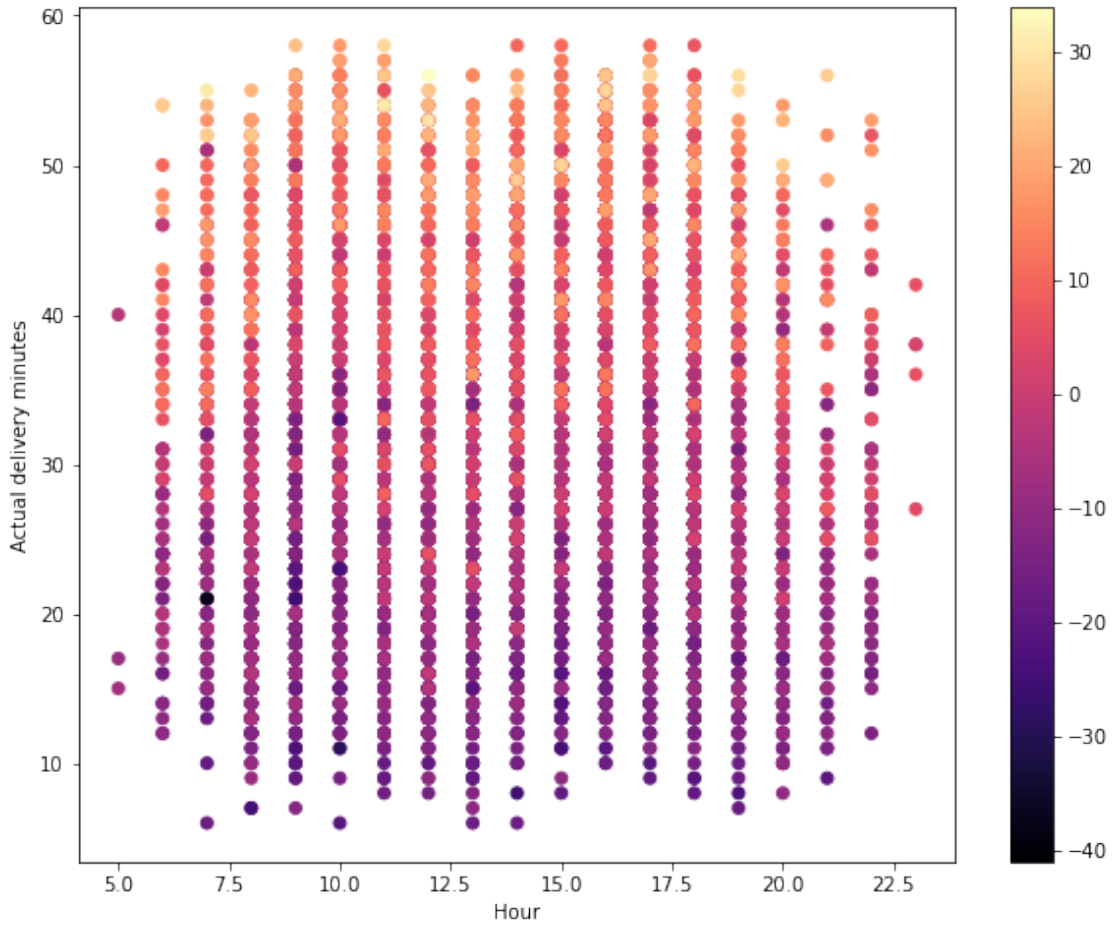
1.0.2 Visualizations

Scatterplots Here we plot distance against actual delivery minutes. Colors describe whether order was late (lighter colors) or early (darker colors). There doesn't seem to be a self evident relationship between distance and actual delivery minutes, though common sense says that distance is probably the best predictor of delivery times. One thing to note is that it looks like late orders (lighter colored dots) are more common for shorter distances than longer. This is probably due to Wolt also assuming that the delivery shouldn't take long for such a short distance like 1 km, but then something goes wrong with the venue or the driver, and delivery is delayed. Also people are more likely to order from venues close to them than from those far away.

[13]:
```
plt.figure(figsize=(10,8))
plt.scatter(orders['DISTANCE'], orders['ACT_MIN'], c=orders['EST_ERROR'],␣
 ↪cmap='magma')
plt.xlabel("Distance")
plt.ylabel("Actual delivery minutes")
plt.colorbar()
plt.show()
```



[14]:
```
plt.figure(figsize=(10,8))
plt.scatter(orders['HOUR_OF_DAY'], orders['ACT_MIN'], c=orders['EST_ERROR'],␣
↪cmap='magma')
plt.xlabel("Hour")
plt.ylabel("Actual delivery minutes")
plt.colorbar()
plt.show()
```



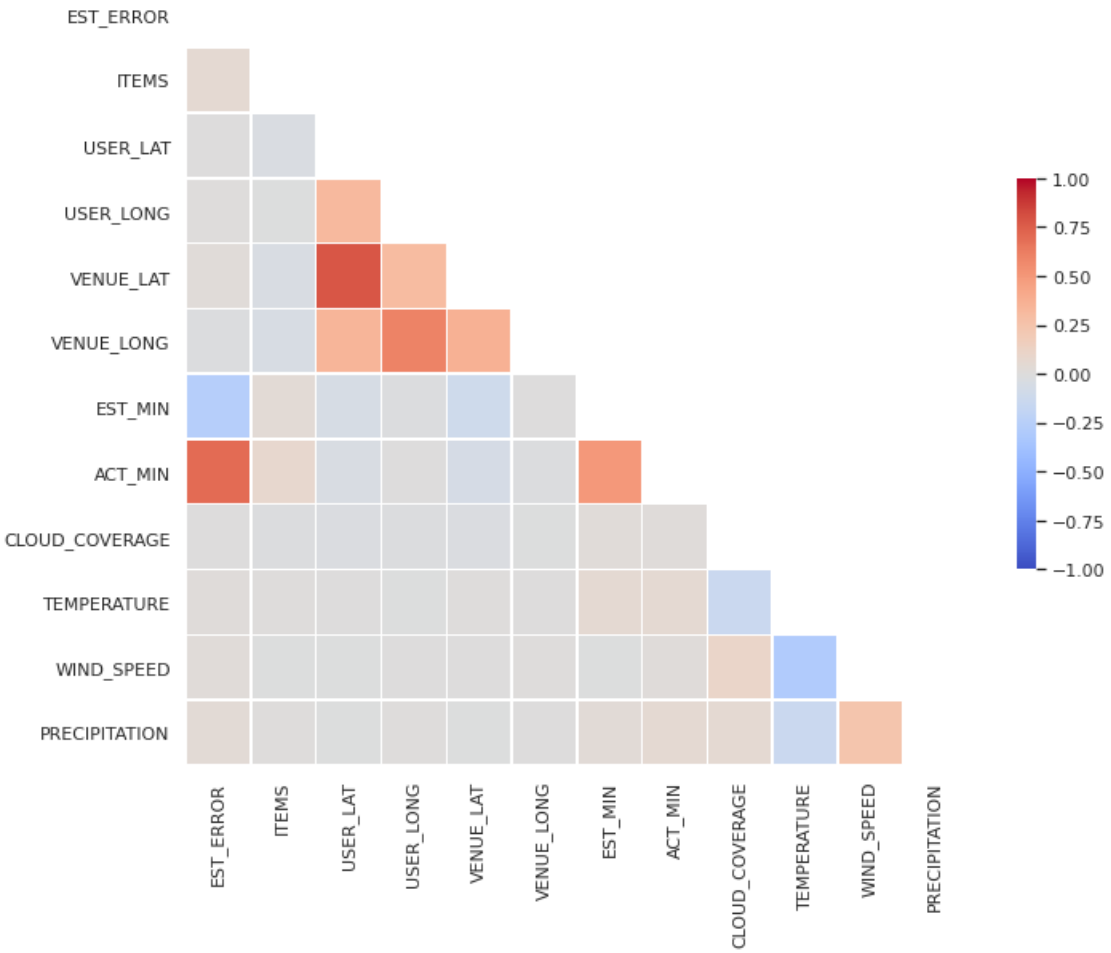
Correlations Correlations of the original variables

[15]:
```
corr = orders.
 ↪drop(columns=['TIME','DATE','WEEKDAY','TIMESTAMP','HOUR_OF_DAY','DISTANCE']).
 ↪corr()
print(corr)
mask = np.triu(np.ones_like(corr, dtype=bool))
sns.set_theme(style="white")
f, ax = plt.subplots(figsize=(11, 9))
cmap = sns.color_palette("coolwarm", as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmin=-1, vmax=1, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

```
                EST_ERROR     ITEMS  USER_LAT  USER_LONG  VENUE_LAT  \
EST_ERROR        1.000000  0.060270  0.007620   0.011212   0.024282
ITEMS            0.060270  1.000000 -0.024591  -0.002642  -0.032220
USER_LAT         0.007620 -0.024591  1.000000   0.327151   0.783507
USER_LONG        0.011212 -0.002642  0.327151   1.000000   0.306335
```

```
VENUE_LAT          0.024282 -0.032220  0.783507   0.306335   1.000000
VENUE_LONG        -0.014073 -0.042992  0.344495   0.604920   0.369177
EST_MIN           -0.258966  0.043399 -0.053572  -0.011572  -0.108297
ACT_MIN            0.706561  0.085816 -0.032420   0.001571  -0.057580
CLOUD_COVERAGE     0.000927 -0.012549 -0.017789  -0.010004  -0.017738
TEMPERATURE        0.019129  0.015198  0.007031  -0.005069   0.008792
WIND_SPEED         0.027668 -0.005275 -0.000166   0.005153   0.002711
PRECIPITATION      0.039914  0.013170 -0.005649   0.009511  -0.007295

                  VENUE_LONG   EST_MIN   ACT_MIN  CLOUD_COVERAGE  TEMPERATURE  \
EST_ERROR          -0.014073 -0.258966  0.706561        0.000927     0.019129
ITEMS              -0.042992  0.043399  0.085816       -0.012549     0.015198
USER_LAT            0.344495 -0.053572 -0.032420       -0.017789     0.007031
USER_LONG           0.604920 -0.011572  0.001571       -0.010004    -0.005069
VENUE_LAT           0.369177 -0.108297 -0.057580       -0.017738     0.008792
VENUE_LONG          1.000000  0.000506 -0.012243       -0.000260     0.000527
EST_MIN             0.000506  1.000000  0.500536        0.025661     0.058621
ACT_MIN            -0.012243  0.500536  1.000000        0.019653     0.060138
CLOUD_COVERAGE     -0.000260  0.025661  0.019653        1.000000    -0.134960
TEMPERATURE         0.000527  0.058621  0.060138       -0.134960     1.000000
WIND_SPEED          0.010771 -0.002554  0.022922        0.104419    -0.291468
PRECIPITATION       0.003050  0.034252  0.060869        0.059631    -0.140488

                  WIND_SPEED  PRECIPITATION
EST_ERROR           0.027668       0.039914
ITEMS              -0.005275       0.013170
USER_LAT           -0.000166      -0.005649
USER_LONG           0.005153       0.009511
VENUE_LAT           0.002711      -0.007295
VENUE_LONG          0.010771       0.003050
EST_MIN            -0.002554       0.034252
ACT_MIN             0.022922       0.060869
CLOUD_COVERAGE      0.104419       0.059631
TEMPERATURE        -0.291468      -0.140488
WIND_SPEED          1.000000       0.244848
PRECIPITATION       0.244848       1.000000
```

[15]: <AxesSubplot:>



Histograms Here is how the actual delivery minutes were distributed:

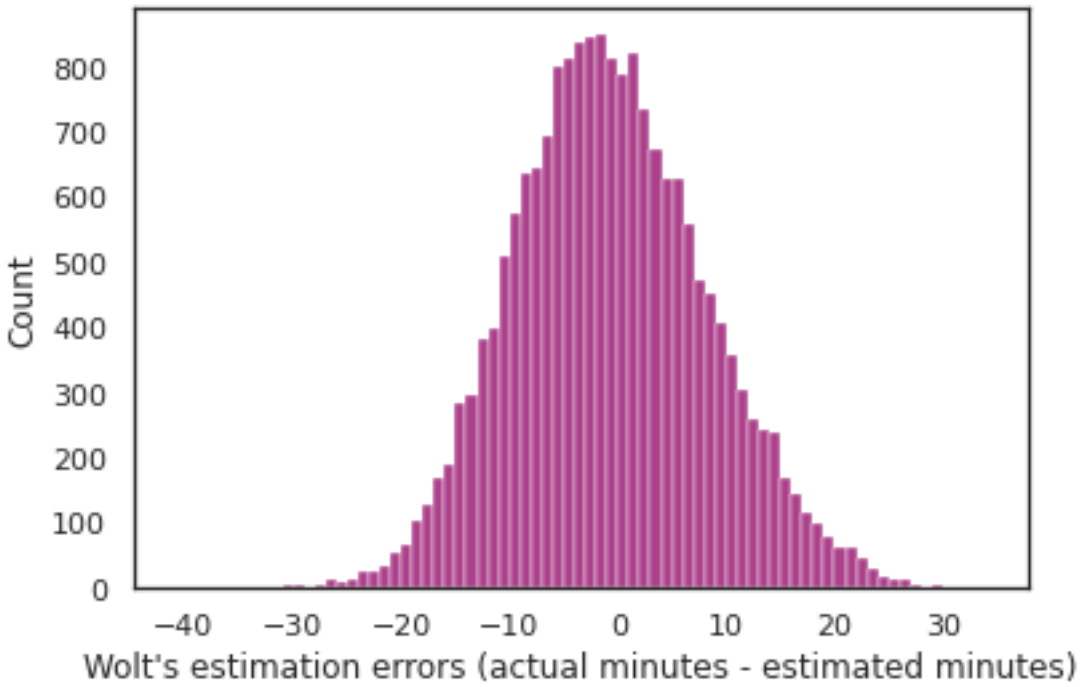
[16]:
```
sns.histplot(data=orders, x='ACT_MIN',
             bins=100, binwidth=1,
             stat='count', color='#910064')
plt.xlabel('Actual delivery minutes')
```

[16]:
```
Text(0.5, 0, 'Actual delivery minutes')
```

Here is how the errors in Wolt's estimated delivery minutes, defined as ACTUAL_DELIVERY_MINUTES - ESTIMATED_DELIVERY_MINUTES, were distributed:

[17]:
```
sns.histplot(data=orders, x='EST_ERROR',
             bins=100,  binwidth=1,
             stat='count', color='#910064')
plt.xlabel("Wolt's estimation errors (actual minutes - estimated minutes)")
```

[17]:
```
Text(0.5, 0, "Wolt's estimation errors (actual minutes - estimated minutes)")
```



There are very few clear relationships between variables except the obvious ones. I have therefore decided to do feature engineering in the hopes that it will give me features that are actually useful in predicting delivery minutes.

1.0.3 Dropping unnecessary columns

To predict delivery minutes (label), I want to use as features distance, the day of the week, and the time of day. At this point, I'm going to drop unnecessary variables from the dataframe.

[18]:
```
df = orders.loc[:,['ACT_MIN','DISTANCE','ITEMS','WEEKDAY','HOUR_OF_DAY']]
df.head()
```

[18]:

	ACT_MIN	DISTANCE	ITEMS	WEEKDAY	HOUR_OF_DAY
0	10	3.881435	1	Saturday	7
1	32	940.839065	8	Saturday	7
2	6	110.668634	4	Saturday	7
3	26	2377.147170	3	Saturday	8
4	26	331.858776	2	Saturday	8

1.0.4 One-hot encoding weekday

[19]:
```
df = pd.concat([df, pd.get_dummies(df['WEEKDAY'])], axis=1)
df
```

[19]:

	ACT_MIN	DISTANCE	ITEMS	WEEKDAY	HOUR_OF_DAY	Friday	Monday	\
0	10	3.881435	1	Saturday	7	0	0	
1	32	940.839065	8	Saturday	7	0	0	
2	6	110.668634	4	Saturday	7	0	0	
3	26	2377.147170	3	Saturday	8	0	0	
4	26	331.858776	2	Saturday	8	0	0	
…	…	…	…	…	…	…	…	
18701	23	55.311498	1	Wednesday	20	0	0	
18702	15	276.794254	6	Wednesday	20	0	0	
18703	11	386.999589	3	Wednesday	20	0	0	
18704	10	55.853594	3	Wednesday	20	0	0	
18705	21	774.957476	1	Wednesday	21	0	0	

	Saturday	Sunday	Thursday	Tuesday	Wednesday
0	1	0	0	0	0
1	1	0	0	0	0
2	1	0	0	0	0
3	1	0	0	0	0
4	1	0	0	0	0
…	…	…	…	…	…
18701	0	0	0	0	1
18702	0	0	0	0	1
18703	0	0	0	0	1
18704	0	0	0	0	1
18705	0	0	0	0	1

[18706 rows x 12 columns]

1.1 Features and label

[20]:
```
df = df.drop(columns=['WEEKDAY'])
df.columns = ['Delivery minutes','Distance','Items','Hour',
 ↪'Friday','Monday','Saturday','Sunday','Thursday','Tuesday','Wednesday']
df = df.reindex(columns=['Delivery minutes','Distance','Items','Hour',
 ↪'Monday','Tuesday','Wednesday','Thursday','Friday','Saturday', 'Sunday'])
df
```

[20]:

	Delivery minutes	Distance	Items	Hour	Monday	Tuesday	Wednesday	\
0	10	3.881435	1	7	0	0	0	
1	32	940.839065	8	7	0	0	0	
2	6	110.668634	4	7	0	0	0	
3	26	2377.147170	3	8	0	0	0	
4	26	331.858776	2	8	0	0	0	
…	…	…	…	…	…	…	…	
18701	23	55.311498	1	20	0	0	1	

```
18702                15   276.794254       6     20        0        0          1
18703                11   386.999589       3     20        0        0          1
18704                10    55.853594       3     20        0        0          1
18705                21   774.957476       1     21        0        0          1

       Thursday  Friday  Saturday  Sunday
0             0       0         1       0
1             0       0         1       0
2             0       0         1       0
3             0       0         1       0
4             0       0         1       0
…           …       …         …       …
18701         0       0         0       0
18702         0       0         0       0
18703         0       0         0       0
18704         0       0         0       0
18705         0       0         0       0

[18706 rows x 11 columns]
```

[21]:
```
X = df.drop(columns=['Delivery minutes']).to_numpy() # All weekdays, time of␣
 ↪day, and distance as features
y = df['Delivery minutes'].to_numpy()

print(X.shape)
print(y.shape)
```

```
(18706, 10)
(18706,)
```

1.2 Benchmark - how good were Wolt's predictions?

We look at MSE and shares of orders that were not too late or early

[22]:
```
mean_squared_error(orders['EST_MIN'],orders['ACT_MIN'])
```

[22]:
```
82.07564417833851
```

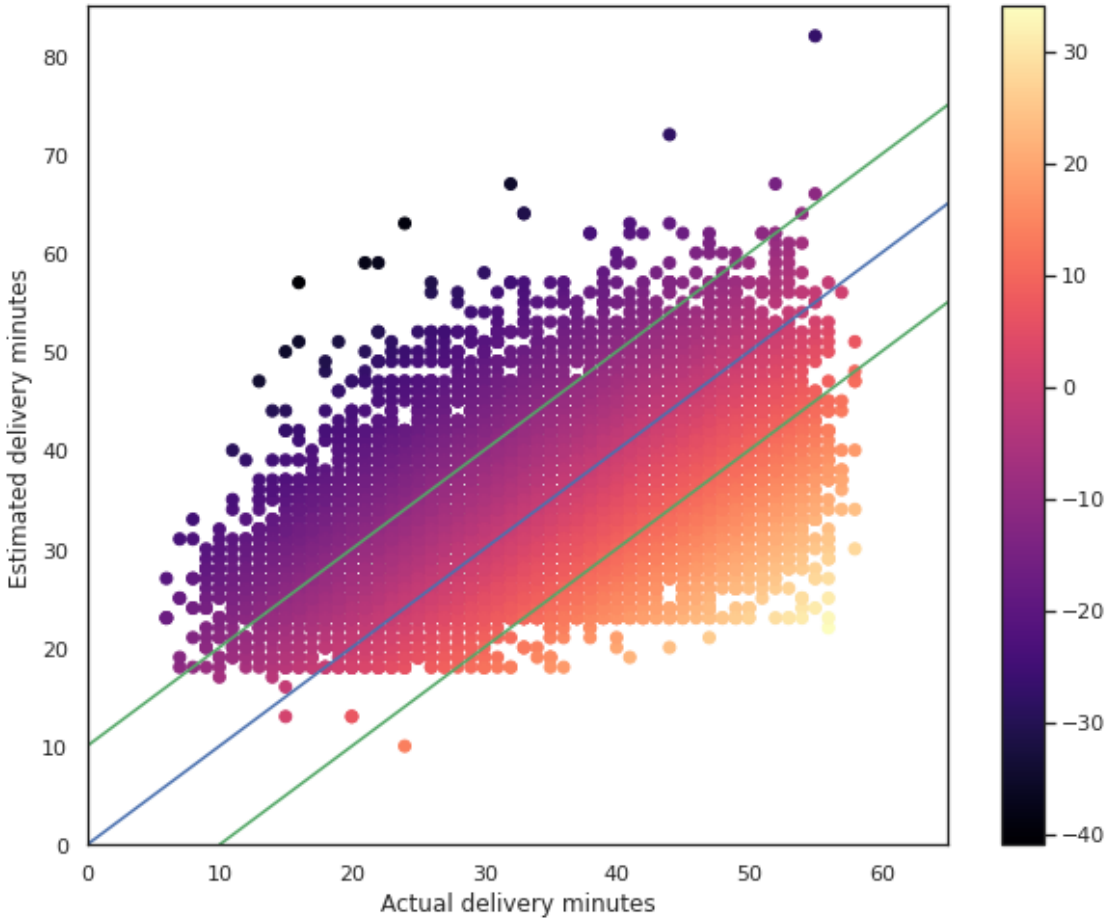
[23]:
```
on_time = sum(np.where(orders['EST_ERROR']==0, 1, 0))
within_5_min = sum(np.where(abs(orders['EST_ERROR'])<=5, 1, 0))
within_10_min = sum(np.where(abs(orders['EST_ERROR'])<=10, 1, 0))

print(f"{on_time} out of Wolt's {len(orders)} orders were exactly on time. So␣
 ↪{round(on_time/len(orders)*100,1)} % of orders were delivered on time.")
print(f"{within_5_min} out of Wolt's {len(orders)} orders were ordered within 5␣
 ↪minutes of the estimated time. So {round(within_5_min/len(orders)*100,1)} %␣
 ↪of orders were delivered within 5 minutes of estimated delivery.")
```

```
print(f"{within_10_min} out of Wolt's {len(orders)} orders were ordered within␣
 ↪10 minutes of the estimated time. So {round(within_10_min/
 ↪len(orders)*100,1)} % of orders were delivered within 10 minutes of␣
 ↪estimated delivery.")
```

```
789 out of Wolt's 18706 orders were exactly on time. So 4.2 % of orders were
delivered on time.
8440 out of Wolt's 18706 orders were ordered within 5 minutes of the estimated
time. So 45.1 % of orders were delivered within 5 minutes of estimated delivery.
14044 out of Wolt's 18706 orders were ordered within 10 minutes of the estimated
time. So 75.1 % of orders were delivered within 10 minutes of estimated
delivery.
```

Here we plot actual delivery minutes against estimated delivery minutes. Colors describe whether order was late (lighter colors) or early (darker colors). The blue line depicts orders that were exactly on time. The area between the two green lines contains orders which were at most 10 minutes early or late. Therefore all data points outside the area depict orders for which Wolt's estimated delivery minutes was off by over 10 minutes. When looking at the scatter this way, we see that Wolt's predictions are often not very accurate. It is difficult to give accurate predictions.

[24]:
```
plt.figure(figsize=(10,8))
plt.scatter(orders['ACT_MIN'], orders['EST_MIN'], c=orders['EST_ERROR'],␣
 ↪cmap='magma')
plt.xlabel("Actual delivery minutes")
plt.ylabel("Estimated delivery minutes")
plt.colorbar()
plt.plot([0, 80], [0, 80], color = 'b')
plt.plot([0, 80], [10, 90], color = 'g')
plt.plot([10, 70], [0, 60], color = 'g')
plt.xlim(0, 65)
plt.ylim(0, 85)
plt.show()
```



2 Models

Split into training, test, and validation data.

[25]:
```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.
↪2, train_size=0.8, random_state=66)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,␣
↪test_size=0.25, train_size=0.75, random_state=66)
```

2.1 Lasso

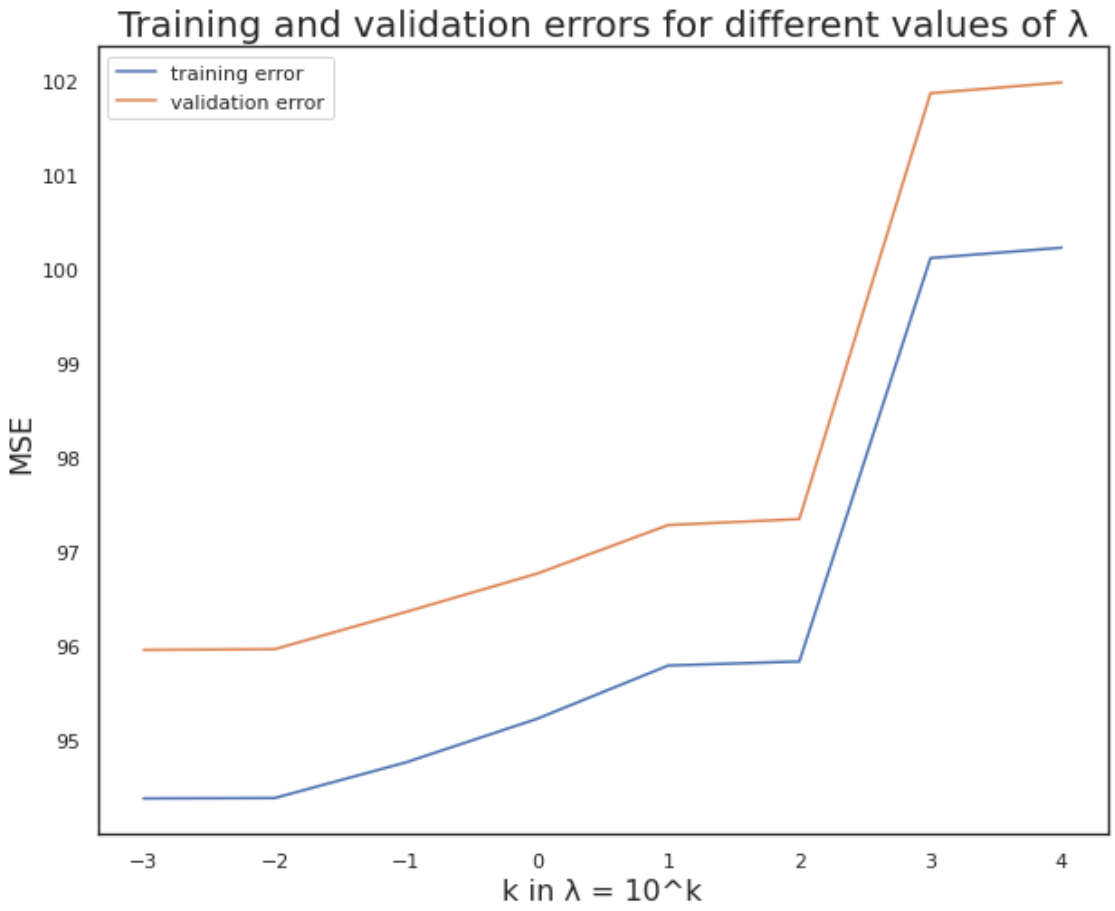
2.1.1 Train the model

[26]:
```
# we will use this variables to store the resulting training/validation errors␣
↪for each polynomial degree
# NB - this time we have multiple errors (for each CV step) for each degree, so␣
↪we store the errors in a dictionary
```

```
tr_errors = []                                  # for recording training erros for␣
 ↪different values of
val_errors = []                                 # for recording validation erros for␣
 ↪different values of
coefs = np.empty((10,12),dtype=float)      # for saving the coefficients and␣
 ↪intercept corresponding to different values of

for i in range(-3,5,1):

    k = 10**i
    # Fit model
    regr = Lasso(alpha=k,fit_intercept=True)
    regr.fit(X_train, y_train)

    # Now we compute the errors on train and validation data obtained from kfold
    y_pred_train = regr.predict(X_train)     # predict using the linear model
    tr_error = mean_squared_error(y_train, y_pred_train)    # calculate the␣
 ↪training error
    y_pred_val = regr.predict(X_val) # predict labels for the validation data␣
 ↪using the linear model
    val_error = mean_squared_error(y_val, y_pred_val) # calculate the␣
 ↪validation error

    # Save errors
    tr_errors.append(tr_error)          # We save the training error for tree␣
 ↪depth = k
    val_errors.append(val_error)        # We save the validation error for tree␣
 ↪depth = k

print("( , training error):")
print(list(zip([10**x for x in range(-3,5,1)],tr_errors)))
print()
print("( , validation error):")
print(list(zip([10**x for x in range(-3,5,1)],val_errors)))
```

```
( , training error):
[(0.001, 94.37983590878929), (0.01, 94.38405259178187), (0.1,
94.76224603390698), (1, 95.22422360941316), (10, 95.78983008130514), (100,
95.83262422107273), (1000, 100.11203819783343), (10000, 100.22214692592507)]

( , validation error):
[(0.001, 95.95536769726675), (0.01, 95.96311143927511), (0.1,
96.35839267821251), (1, 96.76438575079442), (10, 97.27997105811411), (100,
97.34333607850674), (1000, 101.86119924307283), (10000, 101.97508558741303)]
```

[27]:
```
plt.figure(figsize=(10,8))
plt.plot(range(-3,5,1),tr_errors, label="training error")
plt.plot(range(-3,5,1),val_errors, label="validation error")
plt.legend(loc="best")    # set the location of the legend
plt.ylabel('MSE',fontsize=16)
plt.xlabel('k in  = 10^k',fontsize=16)
plt.title(f'Training and validation errors for different values of␣
 ↪ ',fontsize=20)    # set the title
plt.show()
```



Best scaling factor is $\lambda = 0.001$. We use it to compare the model's performance with test data to Wolt's performance:

[28]:
```
lasso_regr = Lasso(alpha=1e-3)
lasso_regr.fit(X_train,y_train)
lasso_ypred = lasso_regr.predict(X_test)
lasso_mse = mean_squared_error(y_test,lasso_ypred)
```

[29]:
```
print("The intercept:")
print(lasso_regr.intercept_)
print("The coefficients for the features:")
print(lasso_regr.coef_)
```

```
The intercept:
27.699831032059762
The coefficients for the features:
[ 0.00440909  0.4802083   0.0456001   1.21371442 -0.         -0.28151164
 -0.32580451  1.0572433  -0.66000167  0.98497743]
```

[30]:
```
print(f"MSE for test data: {lasso_mse}")
diff = y_test-lasso_ypred

my_on_time = sum(np.where(diff==0, 1, 0))
my_within_5_min = sum(np.where(abs(diff)<=5, 1, 0))
my_within_10_min = sum(np.where(abs(diff)<=10, 1, 0))

print(f"{my_on_time} out of {len(y_test)} test predictions were exactly on time.
 ↪ So {round(my_on_time/len(y_test)*100,1)} % of orders were on time.")
print(f"{my_within_5_min} out of {len(y_test)} test predictions were ordered␣
 ↪within 5 minutes of the estimated time. So {round(my_within_5_min/
 ↪len(y_test)*100,1)} % of orders were within 5 minutes of estimated delivery.
 ↪")
print(f"{my_within_10_min} out of {len(y_test)} test predictions were ordered␣
 ↪within 10 minutes of the estimated time. So {round(my_within_10_min/
 ↪len(y_test)*100,1)} % of orders were within 10 minutes of estimated delivery.
 ↪")
```

```
MSE for test data: 93.59643786384298
0 out of 3742 test predictions were exactly on time. So 0.0 % of orders were on
time.
1340 out of 3742 test predictions were ordered within 5 minutes of the estimated
time. So 35.8 % of orders were within 5 minutes of estimated delivery.
2506 out of 3742 test predictions were ordered within 10 minutes of the
estimated time. So 67.0 % of orders were within 10 minutes of estimated
delivery.
```

2.2 Decision tree regression

[31]:
```
# we will use this variables to store the resulting training/validation errors␣
 ↪for each polynomial degree
# NB - this time we have multiple errors (for each CV step) for each degree, so␣
 ↪we store the errors in a dictionary

tr_errors = []                              # for recording training erros for␣
 ↪different values of
```

```
val_errors = []                              # for recording validation erros for␣
 ↪different values of
coefs = np.empty((10,12),dtype=float)      # for saving the coefficients and␣
 ↪intercept corresponding to different values of

for k in range(1,14,1):

    # Fit model
    regr = DecisionTreeRegressor(max_depth=k, criterion='mae')
    regr.fit(X_train, y_train)

    # Now we compute the errors on train and validation data obtained from kfold
    y_pred_train = regr.predict(X_train)    # predict using the linear model
    tr_error = mean_squared_error(y_train, y_pred_train)    # calculate the␣
 ↪training error
    y_pred_val = regr.predict(X_val) # predict labels for the validation data␣
 ↪using the linear model
    val_error = mean_squared_error(y_val, y_pred_val) # calculate the␣
 ↪validation error

    # Save errors
    tr_errors.append(tr_error)          # We save the training error for tree␣
 ↪depth = k
    val_errors.append(val_error)        # We save the validation error for tree␣
 ↪depth = k

print("(Max tree depth,  training error):")
print(list(zip([x for x in range(1,14,1)],tr_errors)))
print()
print("(Max tree depth, average validation error):")
print(list(zip([x for x in range(1,14,1)],val_errors)))
```

```
(Max tree depth,  training error):
[(1, 96.51724137931035), (2, 95.28815824645817), (3, 94.02530517686893), (4,
92.5912857524726), (5, 90.17829457364341), (6, 87.96373518667023), (7,
85.59141940657578), (8, 83.59850307404437), (9, 81.67637886483115), (10,
78.8152900294039), (11, 75.21451483560546), (12, 70.28713356500045), (13,
65.91419406575781)]

(Max tree depth, average validation error):
[(1, 98.38599304998664), (2, 98.36567762630312), (3, 97.09649826249665), (4,
96.24057738572574), (5, 95.51583801122695), (6, 95.09937182571505), (7,
94.42080994386528), (8, 94.02499331729484), (9, 96.58025928896016), (10,
100.87476610531944), (11, 103.79129911788291), (12, 107.16285752472601), (13,
112.35197808072708)]
```

[32]:
```
plt.figure(figsize=(10,8))
plt.plot(range(1,14,1),tr_errors, label="training error")
plt.plot(range(1,14,1),val_errors, label="validation error")
plt.legend(loc="best")
plt.ylabel('MSE',fontsize=16)
plt.xlabel('Max tree depth',fontsize=16)
plt.title(f'Training and validation errors',fontsize=20)
plt.show()
```

Best performance with max tree depth of 7. We run the model with the test data and compare with Wolt and Lasso

[33]:
```
tree_regr = DecisionTreeRegressor(max_depth=7, criterion="mae")
tree_regr.fit(X_train,y_train)
tree_ypred = tree_regr.predict(X_test)
tree_mse = mean_squared_error(y_test,tree_ypred)
```

[34]:
```
print(f"MSE with test data: {tree_mse}")
```

```
diff = y_test-tree_ypred

my_on_time = sum(np.where(diff==0, 1, 0))
my_within_5_min = sum(np.where(abs(diff)<=5, 1, 0))
my_within_10_min = sum(np.where(abs(diff)<=10, 1, 0))

print(f"{my_on_time} out of {len(y_test)} test predictions were exactly on time.
 ↪ So {round(my_on_time/len(y_test)*100,1)} % of orders were on time.")
print(f"{my_within_5_min} out of {len(y_test)} test predictions were ordered␣
 ↪within 5 minutes of the estimated time. So {round(my_within_5_min/
 ↪len(y_test)*100,1)} % of orders were within 5 minutes of estimated delivery.
 ↪")
print(f"{my_within_10_min} out of {len(y_test)} test predictions were ordered␣
 ↪within 10 minutes of the estimated time. So {round(my_within_10_min/
 ↪len(y_test)*100,1)} % of orders were within 10 minutes of estimated delivery.
 ↪")
```

```
MSE with test data: 92.35629342597541
148 out of 3742 test predictions were exactly on time. So 4.0 % of orders were
on time.
1536 out of 3742 test predictions were ordered within 5 minutes of the estimated
time. So 41.0 % of orders were within 5 minutes of estimated delivery.
2634 out of 3742 test predictions were ordered within 10 minutes of the
estimated time. So 70.4 % of orders were within 10 minutes of estimated
delivery.
```

[ ]: