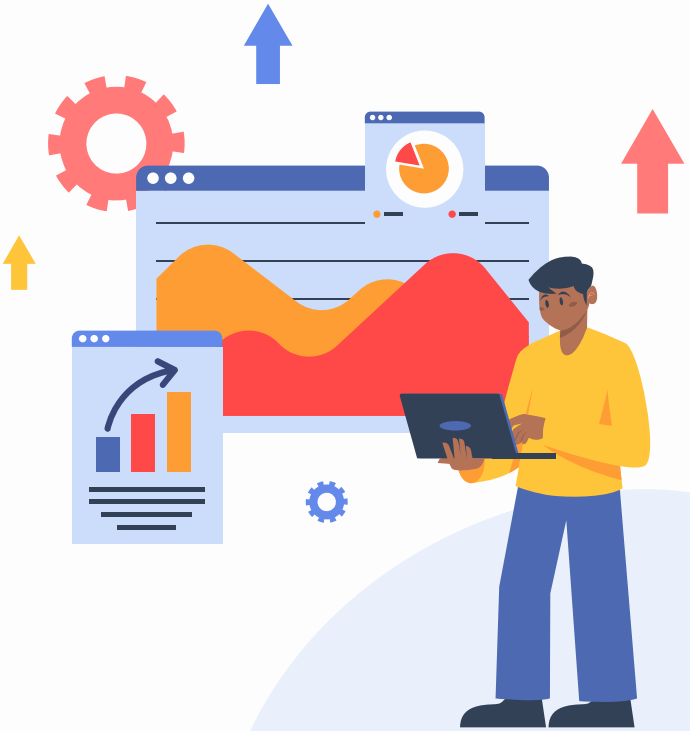
Doordash Prediction Project

By: Abigail Chen, Simran Kaur, Xinchen Li, Sofia Luong



Problem Scope & Data

Problem:

Optimizing the prediction of delivery time

DoorDash ETA Prediction

Motivation:

Help users gain more precise information on their deliveries

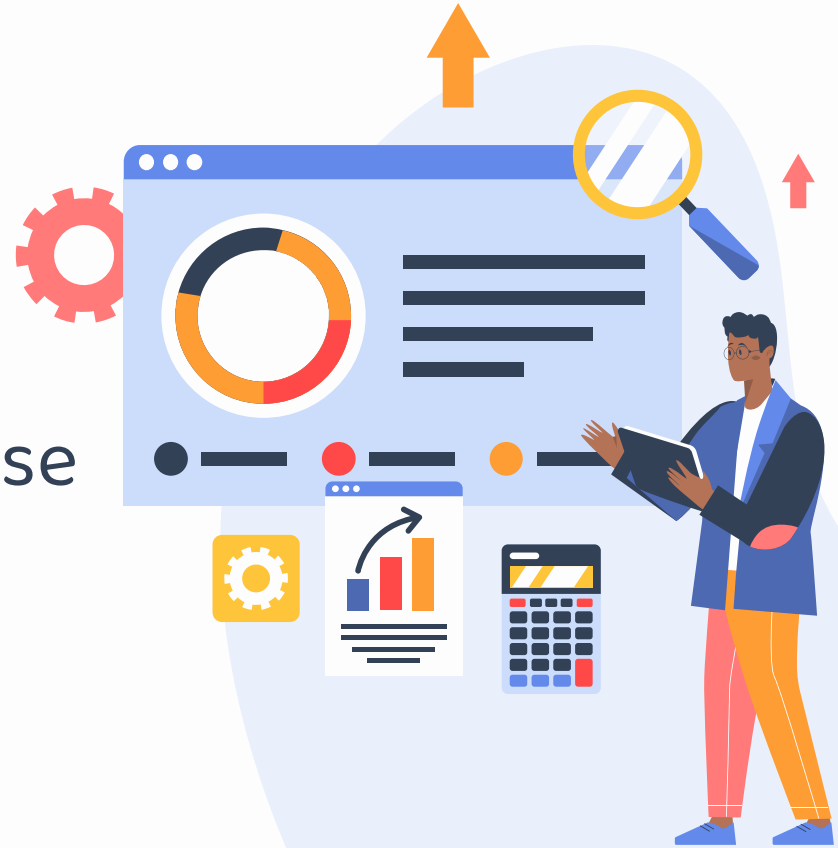
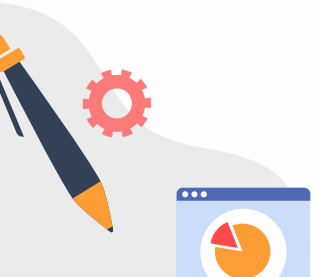


Table of contents



01

Data cleaning, Pre-Processing and EDA

Steps We Took for Preprocessing/Data Cleaning

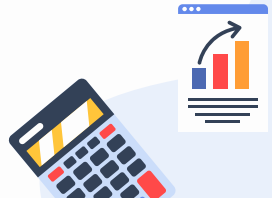
Feature Creation

Dealing with NaN Values

Removing Features

Removing Outliers

Normalizing Data



Feature Engineering - Feature Creation

- **Purpose:**
  - ‘created_at’ & ‘actual_delivery_time’ columns
  - No column for total delivery time (our response variable)

- **Approach:**
  - ‘Total_seconds’ = ‘actual_delivery_time’ - ‘created_at’
  - ‘Weekday’ and ‘hour_of_day’ feature using ‘created_at’
  - Dropped ‘created_at’ and ‘actual_delivery_time’ features

Columns used for Feature Creation

	market_id	created_at	actual_delivery_time	store_id	store_primary_category	order_protocol	total_items	subtotal	num_distinct_items	min_item_price
0	1.0	2015-02-06 22:24:17	2015-02-06 23:27:16	1845	american	1.0	4	3441	4	557
1	2.0	2015-02-10 21:49:25	2015-02-10 22:56:29	5477	mexican	2.0	1	1900	1	1400
2	3.0	2015-01-22 20:39:28	2015-01-22 21:09:09	5477	NaN	1.0	1	1900	1	1900
3	3.0	2015-02-03 21:21:45	2015-02-03 22:13:00	5477	NaN	1.0	6	6900	5	600
4	3.0	2015-02-15 02:40:36	2015-02-15 03:20:26	5477	NaN	1.0	3	3900	3	1100

Code

```
data_cleaned = data.copy()
created_at = pd.to_datetime(data_cleaned['created_at'])
actual_delivery_time = pd.to_datetime(data_cleaned['actual_delivery_time'])

data_cleaned['hour_of_day'] = created_at.dt.hour
data_cleaned['weekday'] = actual_delivery_time.dt.weekday

data_cleaned['total_seconds'] = (actual_delivery_time - created_at).dt.total_seconds()
data_cleaned = data_cleaned[data_cleaned['total_seconds']>0]

data_cleaned = data_cleaned.drop(["actual_delivery_time", "created_at"], axis=1)
```

Dealing with Null values

- **Approach 1 → Dropped rows with missing values:**
  - Columns with large number of missing values
    - ‘total_outstanding_orders’, ‘total_busy_dashers’, ‘total_onshift_dashers
  - Columns with discrete values
    - ‘market_id’ and ‘order_protocol’
- **Approach 2 → Imputed missing values with mean:**
  - Columns with few missing values were imputed with their means.

Null Values

market_id	987
store_id	0
store_primary_category	4760
order_protocol	995
total_items	0
subtotal	0
num_distinct_items	0
min_item_price	0
max_item_price	0
total_onshift_dashers	16262
total_busy_dashers	16262
total_outstanding_orders	16262
estimated_order_place_duration	0
estimated_store_to_consumer_driving_duration	526
hour_of_day	0
weekday	0
total_seconds	0

Too many missing values

Code

```
data_cleaned2 = data_cleaned.dropna(subset=['total_outstanding_orders','total_busy_dashers', 'order_protocol',
                                            'total_onshift_dashers','market_id'])

data_cleaned3 = data_cleaned2.drop(['store_id','store_primary_category'], axis=1)

data_cleaned3 = data_cleaned3.fillna(data_cleaned3.mean())
```

Feature Engineering - Removing Features

- ‘store_primary_category’ was categorical.
  - Too many unique values to one-hot encode
  - low correlation with ‘total_seconds’

- Removed features with extremely high correlations with other predictors
  - 'num_distinct_items'
  - ‘total_onshift_dashers'
  - ‘total_busy_dashers’
  - ‘order_protocol’

Correlation Matrix

	market_id	order_protocol	total_items	subtotal	num_distinct_items	min_item_price	max_item_price	total_onshift_dashers
market_id	1.000000	-0.014876	0.002115	-0.002125	0.013303	-0.009192	-0.005576	0.06946
order_protocol	-0.014876	1.000000	0.005411	-0.052642	-0.026407	-0.041685	-0.088698	0.14520
total_items	0.002115	0.005411	1.000000	0.557509	0.759756	-0.389012	-0.053696	0.03160
subtotal	-0.002125	-0.052642	0.557509	1.000000	0.681519	0.038957	0.508594	0.13093
num_distinct_items	0.013303	-0.026407	0.759756	0.681519	1.000000	-0.445669	0.046610	0.06562
min_item_price	-0.009192	-0.041685	-0.389012	0.038957	-0.445669	1.000000	0.542757	0.04268
max_item_price	-0.005576	-0.088698	-0.053696	0.508594	0.046610	0.542757	1.000000	0.13298
total_onshift_dashers	0.069461	0.145208	0.031607	0.130933	0.065625	0.042681	0.132981	1.00000
total_busy_dashers	0.060163	0.149722	0.028508	0.125688	0.060354	0.044173	0.130909	0.94372
total_outstanding_orders	0.063385	0.135079	0.034295	0.130085	0.067517	0.041207	0.130478	0.93616
estimated_order_place_duration	-0.048696	-0.684553	-0.023991	0.034433	0.003040	0.051891	0.084114	-0.18561
estimated_store_to_consumer_driving_duration	0.017713	-0.010028	0.007443	0.038501	0.025105	0.004138	0.028656	0.04597
hour_of_day	-0.008898	0.013537	-0.070928	-0.190644	-0.119161	-0.051391	-0.188972	-0.37423
weekday	0.000287	0.000454	0.020701	0.029330	0.028227	-0.002020	0.026088	0.10284
total_seconds	-0.038318	-0.047118	0.078685	0.144317	0.106631	0.008554	0.088823	0.04700

Code

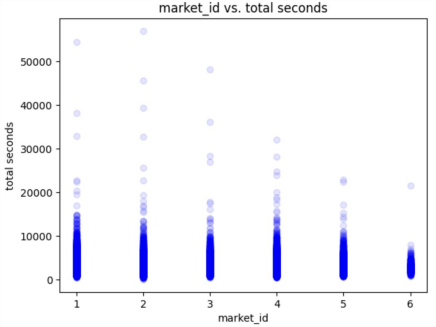
```
data_cleaned3 = data_cleaned2.drop(['store_id','store_primary_category'], axis=1)
```

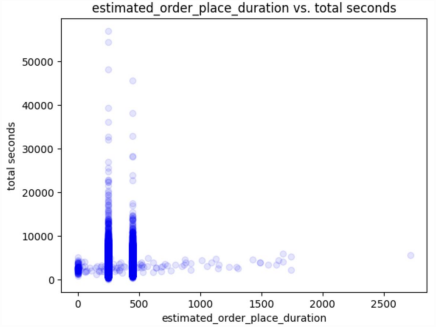
```
data_cleaned4 = data_cleaned3.drop(['num_distinct_items','total_onshift_dashers','total_busy_dashers','order_protocol'], axis=1)
```

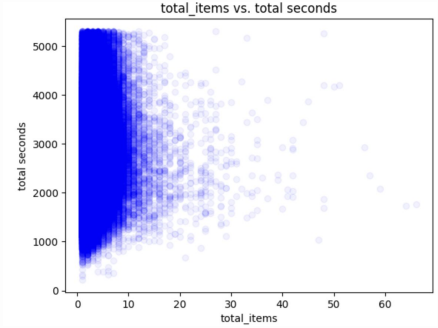
Removing Outliers

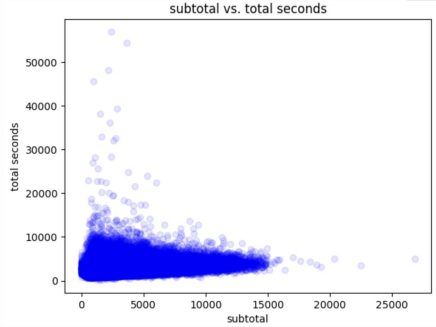
- Plotted scatter plots comparing each feature with the response variable (‘total_seconds’).
- Removed outliers for features that could have a large impact our model’s results
- Removed outliers for:
  - 'total_seconds'
  - 'subtotal'
  - 'estimated_store_to_consumer_driving_duration'
  - 'max_item_price'
  - 'min_item_price'

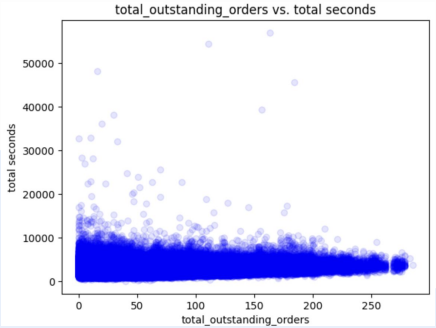
Predictors vs. Response Scatter Plots

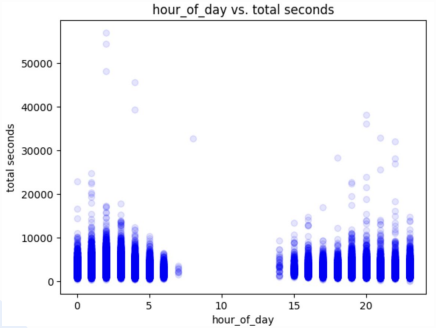


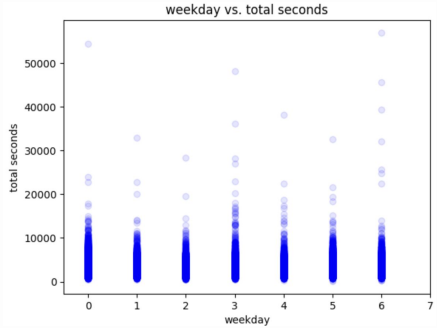


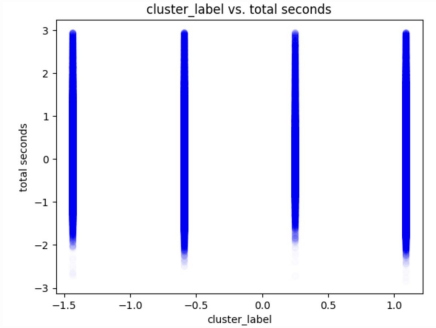




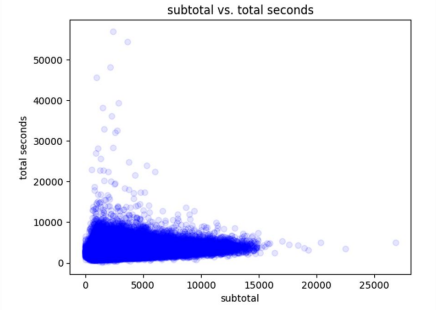


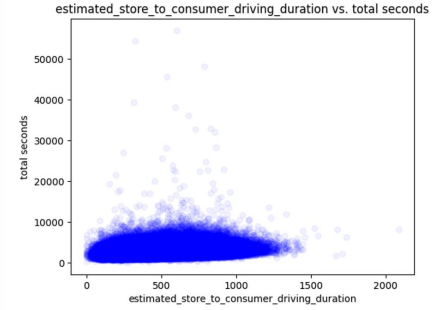


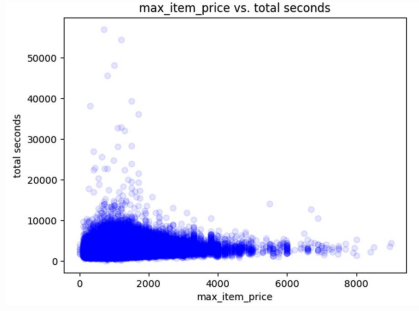


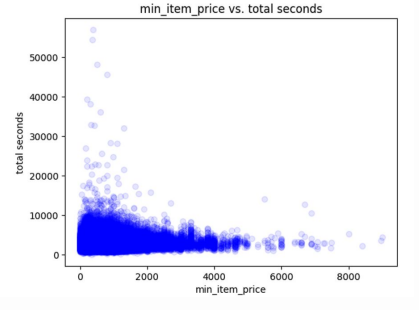


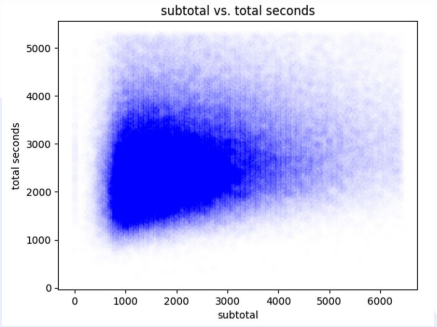
Predictors vs. Response Scatter Plots - Before & After

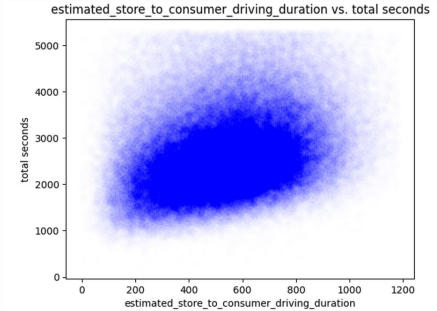


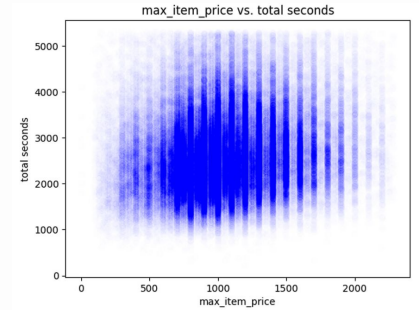


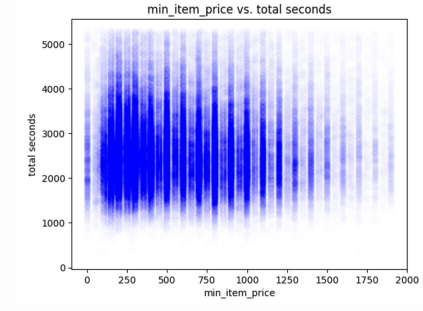












Code

```
df_temp = data_cleaned4

mask = pd.Series(True, index=df_temp.index)

columns = ['total_seconds','subtotal','estimated_store_to_consumer_driving_duration','max_item_price','min_item_price']
for column in columns:
    Q1 = df_temp[column].quantile(0.25)
    Q3 = df_temp[column].quantile(0.75)
    IQR = Q3 - Q1
    in_range = (df_temp[column] >= Q1 - 1.5 * IQR) & (df_temp[column] <= Q3 + 1.5 * IQR)
    mask &= in_range

data_no_outliers = df_temp[mask]

data_no_outliers.describe()
```

Splitting the Data

We used the train_test_split function to split our data into training and testing with a test size of 20%

```
xTrain, xTest, yTrain, yTest = train_test_split(data_no_outliers.drop(['total_seconds'], axis=1),
                                                data_no_outliers['total_seconds'], test_size=0.2, random_state=42)
```

Normalizing Data

- **Purpose:**
  - Across features, numerical range of data was widely different
    - Makes visualization of data more difficult to comprehend
    - Also impacts values of weights + biases
- **Approach:**
  - Used Z-score normalization on the features to bind them in a range

$$Z = \frac{X - \mu}{\sigma}$$

Code

```
scaler = StandardScaler()
xTrain_scaled = pd.DataFrame(scaler.fit_transform(xTrain), columns =  xTrain.columns)
xTest_scaled = pd.DataFrame(scaler.transform(xTest), columns =  xTest.columns)

yTrain_scaled = pd.Series(scaler.fit_transform(yTrain.values.reshape(-1, 1)).ravel())
yTest_scaled = pd.Series(scaler.transform(yTest.values.reshape(-1, 1)).ravel())
```

02

Unsupervised Learning

Clustering

- K-means clustering algorithm → to understand underlying patterns in our data

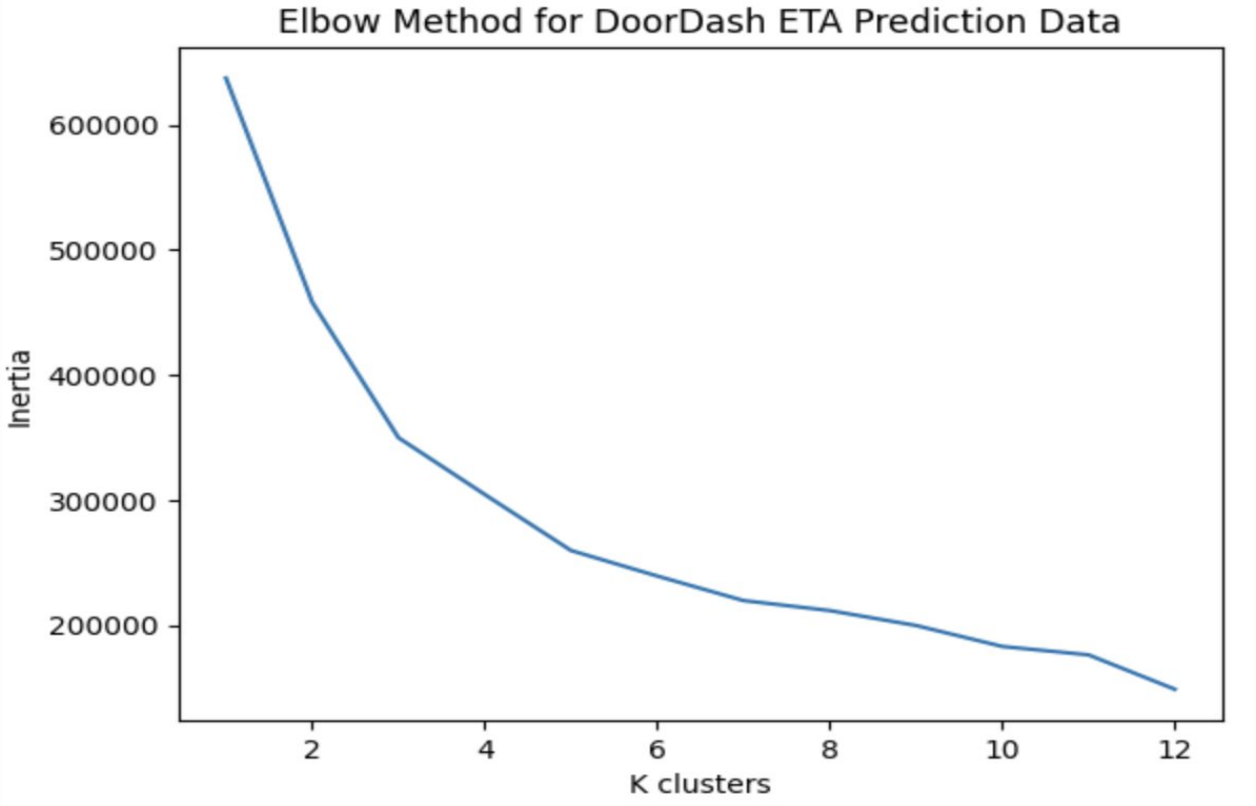
- Methodology for clustering: K-means
  - Used features with clear trend when plotted against the ‘total_seconds’.

  - Used elbow method to find best K-value
    - Tried K-values 1-12
    - Best K-value = 4

Elbow Graph Code

```
# Select K number of centroids
k_scaler = StandardScaler()
scaled_data = pd.DataFrame(k_scaler.fit_transform(data_no_outliers), columns=data_no_outliers.columns)

X_scaled = scaled_data.copy()[['total_items','subtotal','min_item_price','max_item_price']]

# Elbow method to choose K
inertia = []

for k in range(1,13):
  kmeans = KMeans(n_clusters=k, random_state=42)
  clusters = kmeans.fit_predict(X_scaled)
  X_scaled['cluster'] = clusters
  inertia.append(round(kmeans.inertia_, 1))

plt.plot(range(1,13), inertia)
plt.xlabel('K clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method for Heart Data')
plt.show()

#Selected K = 4
```

Elbow Graph


Elbow Method for DoorDash ETA Prediction Data
Inertia
600000
500000
400000
300000
200000
2
4
6
8
10
12
K clusters

Clustering (continued)

- Application
  - **We created a new feature 'cluster_label'** using the class labels obtained from clustered as values
  - Used to fit our Decision Tree and Neural Network models and make predictions

```
kmeans = KMeans(n_clusters=4)
y_kmeans = kmeans.fit_predict(X_scaled)

cluster_labels = kmeans.labels_
scaled_data['cluster_label'] = cluster_labels

print(scaled_data.groupby('cluster_label').mean())

scaled_data['cluster_label'] = k_scaler.fit_transform(scaled_data['cluster_label'].values.reshape(-1, 1))
scaled_data.head()
```

03

Supervised Learning and Methodology

Models constructed

01 **Linear + Polynomial Regression**

02 **Decision Trees**

03 **Neural Networks**

04 **Logistic Regression (ruled out)**

Error metrics: **5-fold Cross Validation for Mean Squared Error (MSE)** and **Mean Absolute Error (MAE)**

Feature Selection - Greedy Algorithm

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

linModel = LinearRegression()
best_metric = float('inf')
selectedPredictors = []
modelTemp = []
columns = list(xTrain.columns)
```

```
for i in range(6):
  best_feature_found = None
  for column in columns:
    if column not in selectedPredictors:
      modelTemp = selectedPredictors + [column]
      linModel.fit(xTrain[modelTemp], yTrain)
      yPred = linModel.predict(xTest[modelTemp])
      mse = mean_squared_error(yTest, yPred)

      if mse < best_metric:
        best_metric = mse
        best_feature_found = column
  if best_feature_found != None:
    selectedPredictors.append(best_feature_found)
  else:
    break

print(selectedPredictors)
xTrain[selectedPredictors].head()
```

Selected predictor variables:

'estimated_store_to_consumer_driving_duration', 'total_outstanding_orders', 'subtotal', 'estimated_order_place_duration', 'hour_of_day', 'market_id'

Linear/Polynomial Regression

- Fitted data to a linear model (degree 1)
- Used polynomial regression to transform it... (degree 2, 3, 4 and 5)
- Applied regularization; MSE and MAE similar to original.
- Used cross-validation; MSE and MAE slightly better.
- Degree 4 is the best polynomial with the lowest MSE and MAE
  - **MSE = 0.8065**
  - **MAE = 0.7120**

```
Degree 1
Original mse: 0.8325774853912852
Original mae: 0.7258654654301407
Ridge mse: 0.8325759910660239
Ridge mae: 0.7258707557910781
cv mse: 0.8387657458135488
cv mae: 0.7282163885849909
Degree 2
Original mse: 0.8132135994526223
Original mae: 0.7167430035193708
Ridge mse: 0.8132099844833449
Ridge mae: 0.716748915130252
cv mse: 0.8183351693750502
cv mae: 0.718362195314951
```

```
Degree 3
Original mse: 0.8046651597150392
Original mae: 0.7129076916913976
Ridge mse: 0.804666798175454
Ridge mae: 0.7129113725097594
cv mse: 0.8100373143909086
cv mae: 0.7149839917191115
Degree 4
Original mse: 0.7991966753348451
Original mae: 0.7102773757185288
Ridge mse: 0.7990076408304524
Ridge mae: 0.7102587901815901
cv mse: 0.8065482230008808
cv mae: 0.7120245876834508
```

```
Degree 5
Original mse: 6.629469077441348
Original mae: 0.7166952463406273
Ridge mse: 0.8270773057130727
Ridge mae: 0.7040410389937369
cv mse: 11.063890741220808
cv mae: 0.7222488319893449
```

Linear/Polynomial Regression (code)

```
for degree in degrees:
  print(f"Degree {degree}")
  # Original Polynomial Regression
  polyreg = make_pipeline(PolynomialFeatures(degree=degree), LinearRegression())
  polyreg.fit(xTrain_scaled[selectedPredictors], yTrain_scaled)
  y_pred = polyreg.predict(xTest_scaled[selectedPredictors])
  # Original MSE and MAE
  mse = mean_squared_error(yTest_scaled, y_pred)
  mae = mean_absolute_error(yTest_scaled, y_pred)
  print(f"Original mse: {mse}")
  print(f"Original mae: {mae}")
```

```
  # Grid search for best alpha
  best_alpha = 1.0
  best_cv_mse = np.inf
  for alpha in alphas:
    polyreg_ridge = make_pipeline(PolynomialFeatu
    cv_mse = -cross_val_score(polyreg_ridge, xTra
    if cv_mse < best_cv_mse:
      best_cv_mse = cv_mse
      best_alpha = alpha
```

```
# Polynomial Regression with Ridge Regularization
polyreg_ridge = make_pipeline(PolynomialFeatures(degree=degree), Ridge(alpha=best_alpha))
polyreg_ridge.fit(xTrain_scaled[selectedPredictors], yTrain_scaled)
y_pred_ridge = polyreg_ridge.predict(xTest_scaled[selectedPredictors])
# Ridge MSE and MAE
mse_ridge = mean_squared_error(yTest_scaled, y_pred_ridge)
mae_ridge = mean_absolute_error(yTest_scaled, y_pred_ridge)
print(f"Ridge mse: {mse_ridge}")
print(f"Ridge mae: {mae_ridge}")
# Cross-Validated MSE and MAE
cv_scores_mse = cross_val_score(polyreg_ridge, xTrain_scaled[selectedPredictors], yTrain_scale
cv_mse = -cv_scores_mse.mean()
cv_scores_mae = cross_val_score(polyreg_ridge, xTrain_scaled[selectedPredictors], yTrain_scale
cv_mae = -cv_scores_mae.mean()
print(f"cv mse: {cv_mse}")
print(f"cv mae: {cv_mae}")
```

Decision Tree

- Deciding on hyperparameters before fitting the data...
  - Implemented a nested for loop to minimize MSE for different hyperparameters
    - Max depth values tested: 10-50
    - Min samples values tested: 100-500 with step = 100
  - Final max depth value: 15
  - Final minimum samples per leaf: 200
- **MSE = 0.7975**
- **MAE = 0.707**

Random Forest

- Hyperparameters
  - Originally started with the same hyperparameters as the decision tree
  - Looked through documentation and played around with other attributes of the object (impurity decrease, bootstrap, warm start, etc.)
    - Didn't perform better than the singular decision tree :(
  - Dropped limitations, set min samples leaf as 50 and left everything else as default
- **MSE = 0.766**
- **MAE = 0.6929**

Decision Tree (code)

Finding best hyperparameters

```
decTreeMSEs = []
decTreeValuesTested = []

for i in range(10, 51):
  for j in range(100, 501, 100):
    decTreeIter = DecisionTreeRegressor(max_depth = i, min_samples_leaf = j, random_state = 0)
    decTreeIter.fit(xTrain_scaled, yTrain_scaled)
    decTreeIterPreds = decTreeIter.predict(xTest_scaled)
    decTreeIterMSE = mean_squared_error(yTest_scaled, decTreeIterPreds)
    decTreeValuesTested += [(i, j)]
    decTreeMSEs += [decTreeIterMSE]

decTreeMinMSE = min(decTreeMSEs)
decTreeMinMSEIndex = decTreeMSEs.index(decTreeMinMSE)
print(decTreeMinMSE, decTreeValuesTested[decTreeMinMSEIndex])
```

```
0.8007229225962674 (15, 200)
```

Decision Tree Fitting & MSE

```
# Decision Tree
from sklearn.tree import DecisionTreeRegressor

decisionTree = DecisionTreeRegressor(max_depth = 15, min_samples_leaf = 200, random_state=0)

decisionTree.fit(xTrain_scaled, yTrain_scaled)
preds = decisionTree.predict(xTest_scaled)

kfold = model_selection.KFold(n_splits=10, random_state=42, shuffle=True)
cvScoreDT = model_selection.cross_val_score(decisionTree, xTrain_scaled, yTrain_scaled, scoring = 'neg_mean_squared_error', cv=kfold)
cvMaeDT = model_selection.cross_val_score(decisionTree, xTrain_scaled, yTrain_scaled, scoring = 'neg_mean_absolute_error', cv=kfold)
```

```
print(f"MSE for Decision Tree: {(-1 * cvScoreDT).mean()}")
print(f"MAE for Decision Tree: {(-1 * cvMaeDT).mean()}")
```

```
MSE for Decision Tree: 0.7959110316323239
MAE for Decision Tree: 0.7070023374030607
```

Random Forest (code)

```python
# Random Forest

randomForest = RandomForestRegressor(n_estimators = 100, min_samples_leaf = 50, random_state = 0)
randomForest.fit(xTrain_scaled, yTrain_scaled)
randomForestPred = randomForest.predict(xTest_scaled)
randomForestScore = mean_squared_error(yTest_scaled, randomForestPred)

kfold = model_selection.KFold(n_splits=10, random_state=42, shuffle=True)
cvScoreRF = model_selection.cross_val_score(randomForest, xTrain_scaled, yTrain_scaled, scoring = 'neg_mean_squared_error', cv=kfold)
cvMaeRF = model_selection.cross_val_score(randomForest, xTrain_scaled, yTrain_scaled, scoring = 'neg_mean_absolute_error', cv=kfold)
```

```python
print(f"MSE for Random Forest: {(-1 * cvScoreRF).mean()}")
print(f"MAE for Random Forest: {(-1 * cvMaeRF).mean()}")
```

```
MSE for Random Forest: 0.7645830826463217
MAE for Random Forest: 0.6929210387572369
```

Neural Networks

- Wanted a more sophisticated model that would (hopefully) capture relationships between predictor and response variables better.
- All predictor variables were used to create the most accurate model possible

- Hyperparameters used:
  - 3 hidden layers with 64, 32, & 16 nodes respectively
  - Activation function = ReLU
  - Lambda for L2 regularization = 0.01
  - 200 epochs

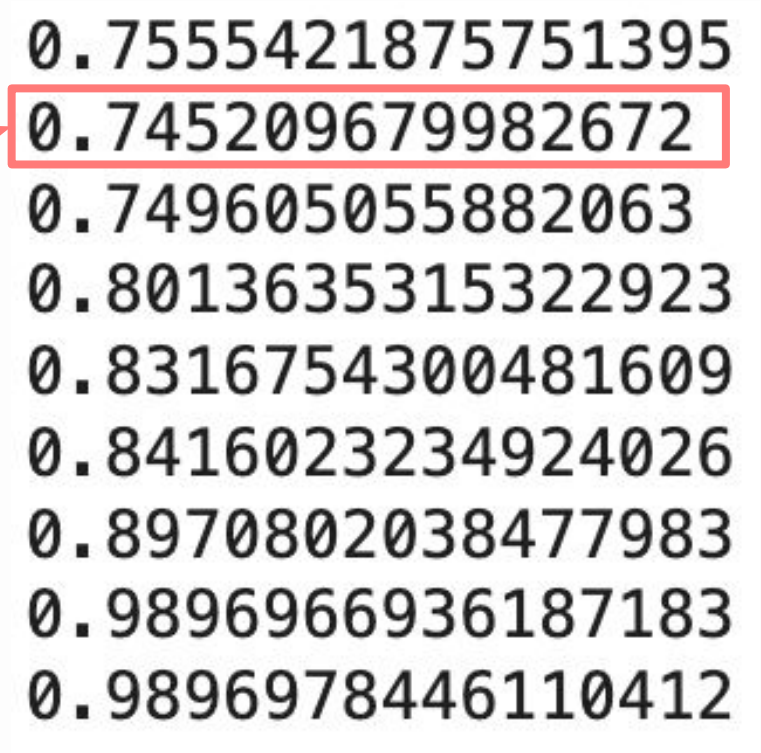
- Results:
  - **MSE = 0.7605**
  - **MAE = 0.6877**

Neural Network - Regularization Term

```
from sklearn.neural_network import MLPRegressor
lambdaList = [0.001, 0.01, 0.1, 1, 5, 10, 20, 100, 1000]

for l in lambdaList:
  NNmodel = MLPRegressor(hidden_layer_sizes=(64,32,16), activation='relu',
                         alpha=l, solver="adam", max_iter=200, random_state=42)

  NNmodel.fit(xTrain_scaled, yTrain_scaled)
  y_pred = NNmodel.predict(xTest_scaled)
  mse = mean_squared_error(yTest_scaled, y_pred)
  print(mse)
```

Neural Network - Regularization Term

0.01 lambda value



Neural Networks (code)

```
kfold = model_selection.KFold(n_splits=10, random_state=42, shuffle=True)
NNmodel = MLPRegressor(hidden_layer_sizes=(64,32,16), activation='relu',
                       alpha=0.1, solver="adam", max_iter=500, random_state=42)

cvScoreNNmse = model_selection.cross_val_score(NNmodel, xTrain_scaled, yTrain_scaled, scoring = 'neg_mean_squared_error', cv=kfold)
cvScoreNNmae = model_selection.cross_val_score(NNmodel, xTrain_scaled, yTrain_scaled, scoring = 'neg_mean_absolute_error', cv=kfold)

print(f"CV Score (MSE): {-1 * cvScoreNNmse}")
print(f"CV Score (MAE): {-1 * cvScoreNNmae}")
print("")
```

```
CV Score (MSE): [0.76493588 0.76572064 0.74217203 0.7560194  0.75387734 0.77964625
 0.77026689 0.77669196 0.8198754  0.74715611]
CV Score (MAE): [0.69121065 0.68810007 0.67996133 0.69268253 0.6962632  0.72870861
 0.6846468  0.67966756 0.69742232 0.68261918]
```

```
print(f"CV-score (MSE) for Neural Network: {(-1 * cvScoreNNmse).mean()}")
print(f"CV-score (MAE) for Neural Network: {(-1 * cvScoreNNmae).mean()}")
```

```
CV-score (MSE) for Neural Network: 0.7605185310574005
CV-score (MAE) for Neural Network: 0.6877246747009738
```

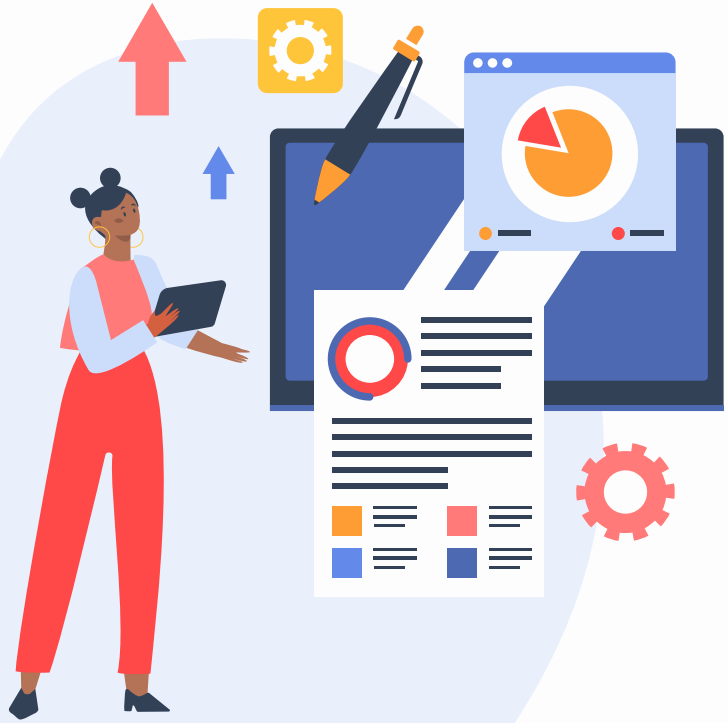
Logistic Regression

- Did **NOT** end up considering logistic regression for final model
- Code on how a logistic regression model could've worked...

```
# Logistic Regression

from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression

logRegModel = LogisticRegression()
logRegModel.fit(xTrain[selectedPredictors], yTrain)
logRegPred = logRegModel.predict(xTest)
logRegKFold = KFold(n_splits = 10, random_state = 5, shuffle = True)
logRegKFoldScore = model_selection.cross_val_score(logRegModel, xTrain[selectedPredictors], yTrain, cv = 5)
print(f"Logistic Regression mean kfold score: {logRegKFoldScore.mean()}")
```

04

Presentation of Results



WHICH MODEL DID WE CHOOSE?

NEURAL NETWORKS

...because it had the best MSE and MAE of all our models

Full List of Error Metrics

Linear Regression

MSE = 0.8390
MAE = 0.728

Polynomial regression

MSE = 0.8073
MAE = 0.7120

Decision Tree

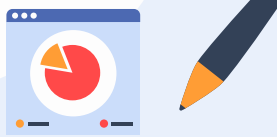
MSE = 0.7974
MAE = 0.7070

Random Forest

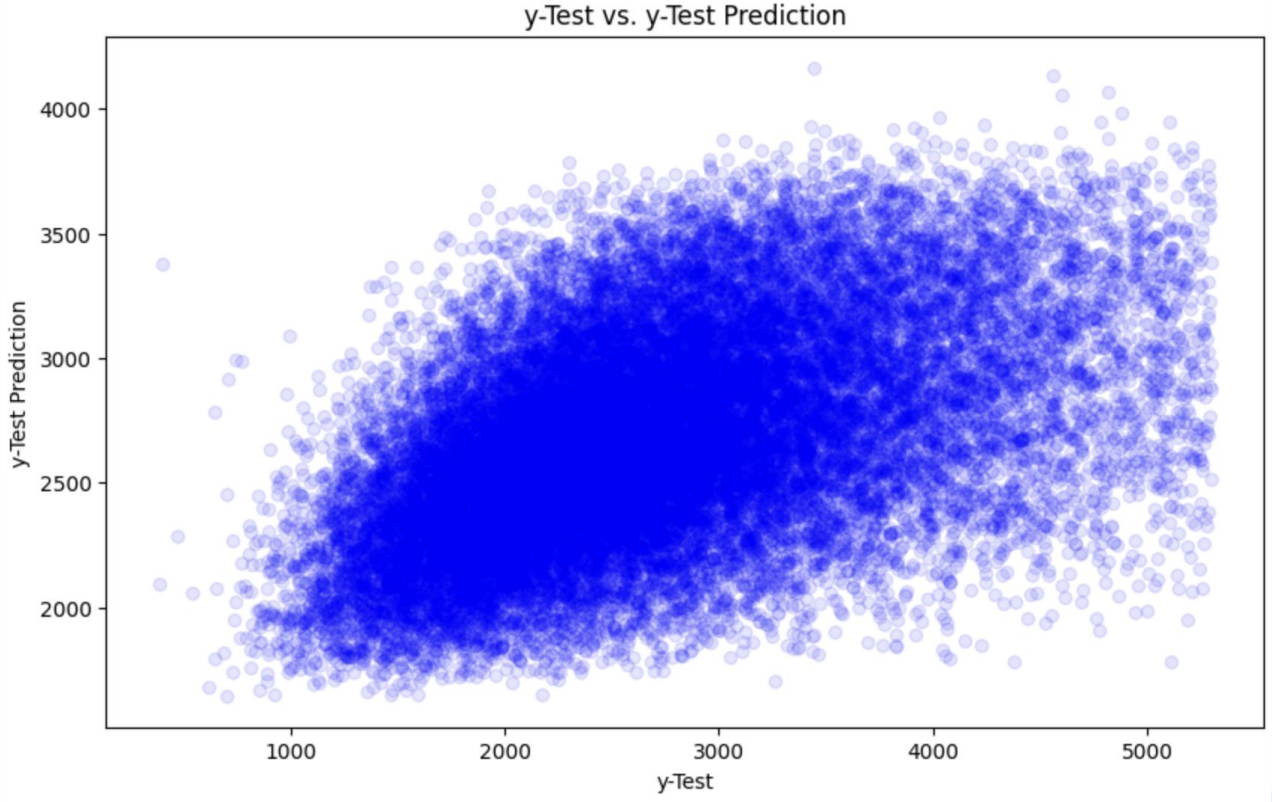
MSE = 0.7660
MAE = 0.6929

Neural Network

MSE = 0.7605
MAE = 0.6877



Y-Test vs. Y-Test Prediction Graph



Summary

- Neural Network Model was used to predict the **total delivery time** of a DoorDash order in seconds

- Used all the predictors (pre-existing and those we created) to generate the best model.

- Why did we choose Neural Networks?
  - Universal function approximators
  - Can discover non-linear relationships to create more accurate predictions.

THANK YOU!!!

(questions?)