به نام خدا

مهدی فقهی

۴۰۱۷۲۲۱۳۶

هوم‌ورک شماره پنج درس یادگیری ماشین

بخش اول :

طبقه بندی بیزی یک روش احتمالی برای طبقه بندی اشیا به یکی از چندین کلاس بر اساس مقادیر مجموعه ای از ویژگی های ورودی است. این روش بر اساس قضیه بیز است، که بیان می کند که احتمال یک فرضیه (در این مورد، یک برچسب کلاس) با توجه به برخی شواهد مشاهده شده (در این مورد، ویژگی های ورودی) با احتمال شواهد ارائه شده به فرضیه متناسب است با ضرب در احتمال قبلی فرضیه. طبقه بندی بیزی در طیف گسترده ای از کاربردها از جمله تشخیص پزشکی، تشخیص تقلب و تشخیص عیب استفاده می شود.

مجموعه داده‌ها: Steel Plates Faults Data Set

ابتدا دیتاست مربوطه را load کردم و بعد از بررسی دادگان آن را به دو دسته آموزش و تست تقسیم کردم ، سپس دادگان آموزش را نرمال کردم و بر حسب پارامترهای بدست آمده از دادگان آموزش، به کمک تابع transform دادگان تست را نیز نرمالیز کردم .

```
# select only the columns containing input features
X_train_features = train_df.iloc[:, :len(input_feature_columns)]

# normalize the input features
scaler = StandardScaler()
model_transfarm = scaler.fit(X_train_features)
train_features_normalized = model_transfarm.transform(X_train_features)

# convert the numpy array of normalized features back into a pandas dataframe
train_features_normalized = pd.DataFrame(train_features_normalized, columns=input_feature_columns)

# combine the input features with other columns in the original training dataframe
# train_normalized = pd.concat([train_features_normalized, train_df.iloc[:, len(input_feature_columns):]], axis=1)

train_normalized = train_features_normalized
```

```
# assume 'df_test' is your input feature DataFrame
test_features = test_df.iloc[:, :len(input_feature_columns)]
test_features_normalized = model_transfarm.transform(test_features)
test_features_normalized = pd.DataFrame(test_features_normalized, columns=input_feature_columns)


# combine the input features with other columns in the original training dataframe
test_normalized = test_features_normalized
```

بعد از انجام این کار سراغ پیاده سازی الگوریتم بیز رفتم .
از آنجا که دادگان ما در این قسمت پیوسته بود به شکل زیر عمل کردم :

برای پیاده سازی شبکه بیز براساس داده‌های پیوسته نیاز داریم که بجای شمارش اتفاق افتادن یک ویژگی در یک برچسب مشخص و پیدا کردن احتمالات آن ، از داده‌های هر ویژگی یک منحنی نرمال بر حسب واریانس و میانگین بسازیم سپس احتمال هر عدد را بر اساس احتمالی که در این منحنی نرمال می‌گیرد بدست می‌آوریم البته به ازای هر کدام از برچسب‌ها برای هر متغیر یک منحنی نرمال می‌سازیم و به ازای هر برچسب از منحنی نرمال همان متغیر استفاده می‌کنیم در نهایت با ضرب این احتمال‌ها براحساس قضیه مارکوف و بزرگی بدست آمده برچسب را پیدا می‌کنیم. از آنجایی که ممکن بود عدد ما خیلی کوچک شود به جای ضرب از لگاریتم احتمال و جمع لگاریتم‌ها استفاده کردیم که همان معنا را می‌دهد همچنین برای اسموت کرد احتمالات از یک آلفا استفاده می‌کنیم که در بدترین حالت‌ها نیز برای یک عدد پیوسته متغیر ما احتمالی را در نظر بگیرد.

```python
class MehNaiveBayes_on:
    def __init__(self, alpha=1):
        self.alpha = alpha

    def fit(self, vectors_feature, y_train, list_label):
        n_samples, n_features = vectors_feature.shape
        self.classes = np.unique(list_label)
        n_classes = len(self.classes)
        self.mean = np.zeros((n_classes, n_features))
        self.var = np.zeros((n_classes, n_features))
        self.priors = np.zeros(n_classes)

        for i, c in enumerate(self.classes):
            X_c = vectors_feature[y_train == c]
            self.mean[i] = X_c.mean(axis=0)
            self.var[i] = X_c.var(axis=0) + self.alpha # Laplace smoothing
            self.priors[i] = X_c.shape[0] / float(n_samples)

    def predict(self, vectors_feature, name_pos, name_neg):
        y_pred = []
        for vect in vectors_feature:
            pos_prob = np.log(self.priors[0])
            neg_prob = np.log(self.priors[1])
            for i in range(vectors_feature.shape[1]):
                x = vect[i]
                if x != 0:
                    pos_prob += np.log(self._pdf(x, self.mean[0][i], self.var[0][i]))
                    neg_prob += np.log(self._pdf(x, self.mean[1][i], self.var[1][i]))
            if pos_prob > neg_prob:
                y_pred.append(name_pos)
            else:
                y_pred.append(name_neg)
        return y_pred

    def _pdf(self, x, mean, var):
        numerator = np.exp(- (x - mean) ** 2 / (2 * var))
        denominator = np.sqrt(2 * np.pi * var)
        return numerator / denominator
```

برای پیدا کردن بهترین میزان لاپلاس بعد از پیاده سازی بالا، به ازای هر کدام از مقادیر لاپلاس دقت را حساب کردیم تا به بهترین میزان آن برای دادگان خود به عنوان یک هایپر پارامتر برسیم .

```python
list_of_alfa = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]

for item in list_of_alfa:
  naive_beyes = MehNaiveBayes_on(item)
  naive_beyes.fit(x_train, y_train,[1,2])
  y_pred = naive_beyes.predict(x_test,1,2)
  score1 = Me_Calcu_Accuracy.accuracy_score(y_test,y_pred)
  print(score1)
```

```
0.9948586118251928
0.9768637532133676
0.8997429305912596
0.8123393316195373
0.7892030848329049
0.781491002570694
0.7686375321336761
0.7506426735218509
0.7249357326478149
0.6940874035989717
```

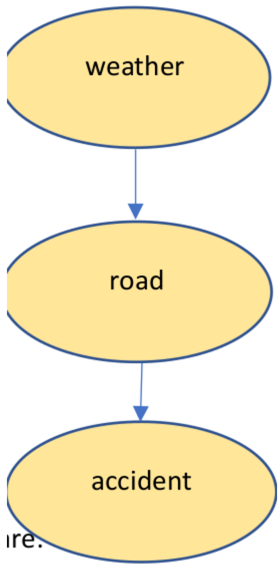
همانطور که مشاهده می فرمایید بهترین میزان آن به ازای 0.1 بدست آمده است حال بقیه معیارها را نیز برحسب این میزان بدست می‌آوریم :

```
precision :  0.9919354838709677

recall :  1.0

f1_measure L  0.995951417004048!
```

confusion matrix :

```
[141, 0]
[2, 246]
```

بخش دوم :



می‌دانیم که به صورت گرافی احتمالات ما به شکل بالا نشان داده می‌شوند برای پیدا کردن هر کدام از احتمالات خواسته شده برای این بخش از کتاب‌خانه pgmpy استفاده کردم .

ابتدا براساس شرایط گفته شده مدل بیز خود را تعریف کردم :

```
# Defining the Bayesian Model Structure
model = BayesianModel([('Weather', 'Road'), ('Road', 'Accident')])

cpd_weather = TabularCPD(variable='Weather', variable_card=2, values=[[0.7], [0.3]])
cpd_road = TabularCPD(variable='Road', variable_card=2,
                  values=[[0.9, 0.6], [0.1, 0.4]],
                  evidence=['Weather'], evidence_card=[2])
cpd_accident = TabularCPD(variable='Accident', variable_card=2,
                  values=[[0.95, 0.25], [0.05, 0.75]],
                  evidence=['Road'], evidence_card=[2])

# Associating the CPDs with the network structure
model.add_cpds(cpd_weather, cpd_road, cpd_accident)
```

سپس به شکل‌زیر خواست سوال را برآورده کردم :

```python
# Task 1: Calculate the probability of a car accident when the weather is sunny and the road
# is dry.
from pgmpy.inference import VariableElimination

# Create an inference object
infer = VariableElimination(model)

# Compute P(Accident | Road=dry, Weather=sunny)
q = infer.query(['Accident'], evidence={'Road':0 , 'Weather': 0})
print(q)
```

حاصل کوئری بالا برابر است با :

```
+-------------+-----------------+
| Accident    |   phi(Accident) |
+=============+=================+
| Accident(0) |          0.9500 |
+-------------+-----------------+
| Accident(1) |          0.0500 |
+-------------+-----------------+
```

که در صورت برقرار بودن شرایط بالا اعلام می کند که احتمال تصادف برابر با 0.05 می‌باشد .

```python
# Task 2: Calculate the probability of a car accident when the weather is rainy and the road
# is wet.

# Compute P(Accident | Road=wet, Weather=rainy)
q = infer.query(['Accident'], evidence={'Road':1 , 'Weather': 1})
print(q)
```

```
+-------------+-----------------+
| Accident    |   phi(Accident) |
+=============+=================+
| Accident(0) |          0.2500 |
+-------------+-----------------+
| Accident(1) |          0.7500 |
+             +                 +
```

در اینجا نیز با برقرار شرایط بارانی بودن و خیس بودن جاده احتمال تصادف برابر با 0.75 درصد است .

```python
# Task 3: Calculate the conditional probability distribution of Accident given Weather=rainy.

q = infer.query(['Accident'], evidence={'Weather': 1})
print(q)
```

```
+-------------+-----------------+
| Accident    |   phi(Accident) |
+=============+=================+
| Accident(0) |          0.6700 |
+-------------+-----------------+
| Accident(1) |          0.3300 |
+-------------+-----------------+
```

در صورتی که هوا بارانی باشد نیز احتمال تصادف برابر با 0.33 درصد است .

بخش سوم :

ابتدا بر اساس سرعت ثابت و مکان اولیه ground truth را محاسبه می کنیم .

```
#lets calculate ground truth
GT = []
x = x0
for i in range(100): #lets calculate it for 100 intervals.
    v0 = v0
    x = x + dt*v0
    GT.append(x)
```

سپس براساس نویز نرمال بر حسب واریانس و میانگین گفته شده مقداری که سنسور اعلام می کند را محاسبه می کنیم .

```
#in this func, with mean = mu and variance of sd
def noise(mu,sd):
    n = np.random.normal(mu,sd)
    return n
```

```
#lets calculate sensor loacations
SL = []
x = x0
v = v0
SV = [] #sensor recorded velocity
SV.append(30)
for i in range(100):
    x_t_1 = x
    x = x + dt*v0 + noise(mu,sd) #here we add random noise to it
    SL.append(x)
    v = (x - x_t_1)/dt
    SV.append(v)
```

حال از kelman filter برای تخمین برحسب مشاهدات سنسور استفاده می کنیم.

```
from pykalman import KalmanFilter

T = 100     # number of time steps
time = np.arange(T) * dt

# Initialize the Kalman filter
kf = KalmanFilter(
    initial_state_mean = [0, v0],
    initial_state_covariance = np.eye(2),
    transition_matrices = [[1, dt], [0, 1]],
    observation_matrices = [[1, 0]],
    observation_covariance = sd**2,
    transition_covariance = np.zeros((2,2)),
)

# Perform the Kalman filtering
(filtered_state_means, filtered_state_covariances) = kf.filter(SL)

# Extract the estimated position from the filtered state means
estimated_position = filtered_state_means[:,0]
```

همانطور که می‌بینیم kalman filter یک initial state می گیره که با توجه به اطلاعات اولیه که در نقطه صفر داریم می‌دانیم از نقطه صفر شروع به حرکت کرده است با سرعت ثابت v0 سپس چون ما مکان اولیه و سرعت اولیه را می‌دانیم پس initail state covariance ما برابر با [1,1] هست که یعنی در شروع ما هیچ نویزی نداریم .

x_nنشان دهنده مکان قطار در زمان n است. سرعت قطار ابه عنوان نرخ تغییردامنه با توجه به زمان تعریف میشود، بنابراین سرعت مشتق از مکان است:

$$\dot{x} = v = \frac{dx}{dt}$$

به صورت کلی میتوان گفت که معادله نیوتن برای بدست آوردن مکان بر اساس سرعت و شتاب به صورت زیر تعریف میشود :

$$X_t = \frac{1}{2} dt*a_{t-1} + V_{t-1}*dt+ X_{t-1}$$

درفرمول فوق a همان شتاب است و V نیز همان سرعت است،حال در مسئله ما سرعت را ثابت در نظر گرفته ایم ،لذا مقدار شتاب برابر با 0 خواهد بود و در نهایت به دو رابطه زیر میرسیم :

$$X_t = V_{t-1}*dt+ X_{t-1}$$

$$V_t = V_{t-1}$$

حال نیاز داریم که این روابط را در قالب ضرب هایماتریسی بدست آوریم، طبق بیان مسئله هر حالت (وضعیت) ما شامل یک جفت از مکان و سرعت میباشد که میتوانیم با در نظر گرفتن $X = x$ و $V = \dot{x}$ خواهیم داشت :

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix} = f \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

که قرار است از مکان و سرعت قبلی به مکان و سرعت جدید برویم، حال باید ماتریس F را به نحوی تعریف کنیم که دو رابطه قبلی‌را با آنها بدست آوریم، لذا خواهیم داشت :

$$f = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix}$$

چرا که با ضرب این ماتریس در بردار مکان -سرعتبه همان روابط فیزیک اولیه می‌رسیم

$$fx = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 1 \times x + dt \times \dot{x} \\ 0 \times x + 1 \times \dot{x} \end{bmatrix}$$

:

درادامه نیاز به بدست آوردن مقدار اندازه گیری سنسورداریم، از آنجا که سنسور صرفا مکان را اندازه میگیرد،ما باید از رابطه زیراستفاده کنیم و در آن رابطه H را به نحوی تعریف کنیم که صرفا مقدار مکان را بدست آورد و مقدار سرعت را بدست نیاورد پس خواهیم داشت :

$$z = Hx$$

$$H = [1,0]$$

پس برهمین اساس :

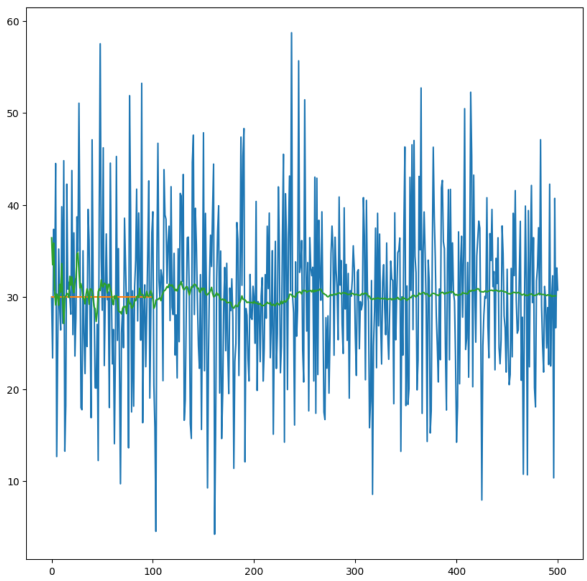
```
transition_matrices = [[1, dt], [0, 1]],
observation_matrices = [[1, 0]],
```

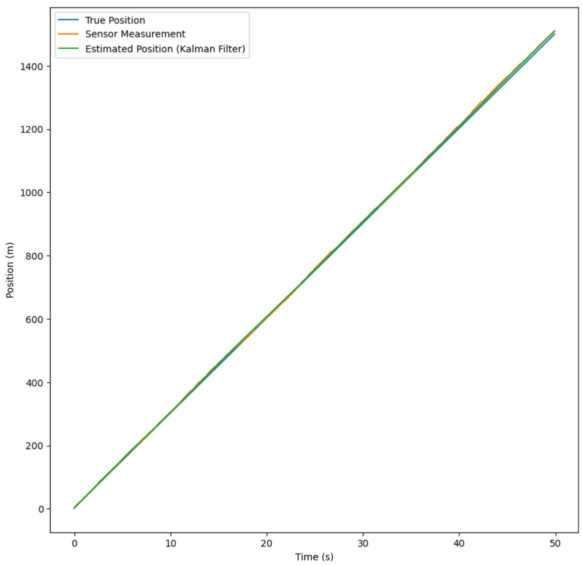
نویز انتقال وضعیت‌در مسئله ما، مقدار نویزی از این بابت نداریم، چرا که با وجود سرعت ثابت و روابط دقیق نیوتن، می توانیم بدون وجود نویز مقادیر بعدی را بدست آوریم،لذا این ماتریس را به صورت زیر تعریف میکنیم :

$$\Sigma_x = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

پس خواهیم داشت :

```
transition_covariance = np.zeros((2,2))
```





در بالا نمودار تخمین مکان با نویز و بدون نویز و نتیجه حاصل از فیلتر و همچنین تخمین سرعت را در چپ بر حسب مکان می‌بینید .

v0 (سرعت)	sd (انحراف معیار)	mse without filter	mse with filter
30	0.1	0.53	0.78
30	0.3	0.72	1.05
30	0.6	5.3	7.6
30	0.9	13.46	21.14
60	0.1	0.2	0.21
60	0.3	2	5.7
60	0.6	4	12
60	0.9	10	19
90	0.1	0.33	0.44
90	0.3	0.47	0.86
90	0.6	4.2	9.8
90	0.9	28.56	30.2

باتوجه به محاسبات فوق که انجام دادیم به این نتیجه میرسیم که سرعت اولیه ثابت ما آنقدر تاثیر ندارد و احتمالا تفاوت در مقدارنوسانات ما به علت وجود مقدار نویز های تصادفی باشد، ولی از طرفی مشاهده کردیم که با افزایش میزان نویز،مقدار خطای ما نیز افزایش پیدا میکند و با کاهش آن مقدار ضرر ما کاهش پیدا میکند.

درآخر نیز بیان شده است که اگر مقدار اولیه مکان را متفاوت در نظر بگیریم،چه میشود، اگر ما بیاییم و صرفا در قسمت سنسور این مقدار را تغییر دهیم، رویدادی که تجربه میکنیم این است که نمودار را کمی به سمت بالا یا پایین شیفت داده خواهد شد . بالا یا پایین بر اساس مقدار کاهش یا افزایش مکان یا سرعت.

بخش چهارم

ابتدا مانند پروژه قبل grand truth را مکان، سرعت و شتاب بدست می‌آوریم .

مطمئناً، با توجه به اینکه موقعیت سر فنر متغیر حالت مورد نظر است، می‌توانیم از یک مدل انتقال حالت استفاده کنیم که رابطه بین موقعیت فعلی «x_t»، سرعت «v_t» و شتاب «a_t» را با موقعیت قبلی نشان می‌دهد.

x_t = x_{t-1} + v_{t-1} * dt + 0.5 * a_{t-1} * dt^2

v_t = v_{t-1} + a_{t-1} * dt

a_t = f(x_t

```
# Define the true position, velocity, and acceleration of the spring head
x_true = np.sin(time)
v_true = np.cos(time)
a_true = -k * x_true / m
```

a_t = -k * x_t

سپس مقادیر بالا را برحسب اینکه مکان را به صورت نویزی سنسور بدست آورد بدست می‌آوریم پس داریم :

```
z = x_true + np.random.normal(0, sensor_noise, size=T)
z_v =[0]
for i in range(len(z)-1):
    z_vel = (z[i+1] - z[i])/dt
    z_v.append(z_vel)

z_a = [0]
for i in range(len(z_v)-1):
    z_ace = (z_v[i+1] - z_v[i])/dt
    z_a.append(z_ace)
```

سپس از فیلتر kelman برای تخمین بهتر استفاده می کنیم.

```
# Initialize the Kalman filter
kf = KalmanFilter(
    initial_state_mean=[0, 0, 0],
    initial_state_covariance=np.eye(3),
    transition_matrices=[[1, dt, 0.5*dt**2],
                         [0, 1, dt],
                         [0, -k/m*dt, 1]],
    observation_matrices=[[1, 0, 0]],
    observation_covariance=sensor_noise**2,
    transition_covariance=np.diag([0, 0.1, 1])**2,
    em_vars=['transition_covariance', 'observation_covariance']
)

# Perform the Kalman filtering on the noisy sensor measurements
(filtered_state_means, filtered_state_covariances) = kf.filter(z)
```

در اینجا نیز داریم :

در این برنامه ، ابتدا پارامترهای سیستم شامل ثابت فنر، جرم، گام زمانی و تعداد مراحل زمانی را تعریف می کنیم. سپس موقعیت، سرعت و شتاب واقعی سر فنر را با فرض یک حرکت سینوسی تولید می‌کنیم و اندازه‌گیری سنسور نویز موقعیت را با اضافه کردن نویز گاوسی به موقعیت واقعی محاسبه می‌کنیم. سپس، فیلتر کالمن را با میانگین حالت اولیه «[0، 0، 0]» مقداردهی اولیه می‌کنیم، که موقعیت، سرعت و شتاب صفر و یک کوواریانس حالت اولیه ماتریس هویت را نشان می‌دهد. ماتریس‌های انتقال، ماتریس‌های مشاهده و کوواریانس‌ها را با استفاده از مدل انتقالی که قبلاً تعریف شد، مشخص می‌کنیم و به فیلتر کالمن اجازه می‌دهیم تا کوواریانس‌های انتقال و مشاهده را با استفاده از تخمین حداکثر احتمال تخمین بزند.

براساس فرمولیشن کشش فنر می‌دانیم که باید F به گونه زیر تعریف شود :

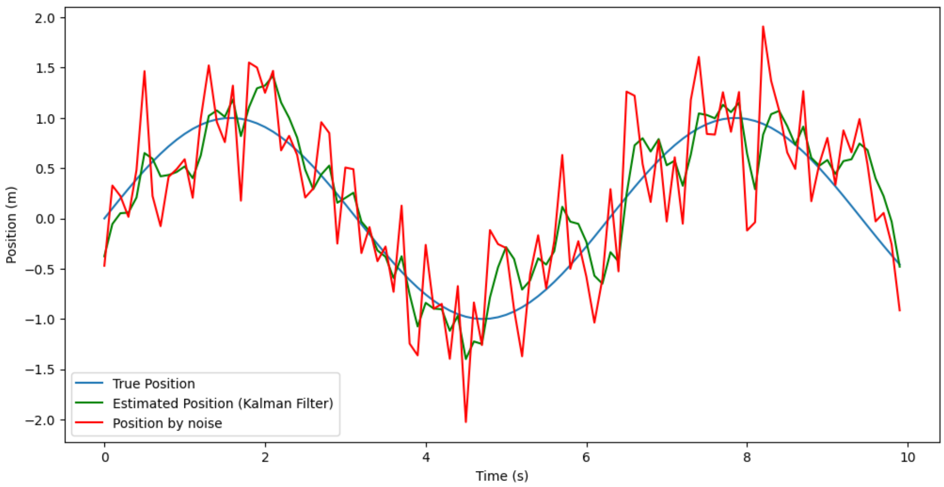
$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & .5\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

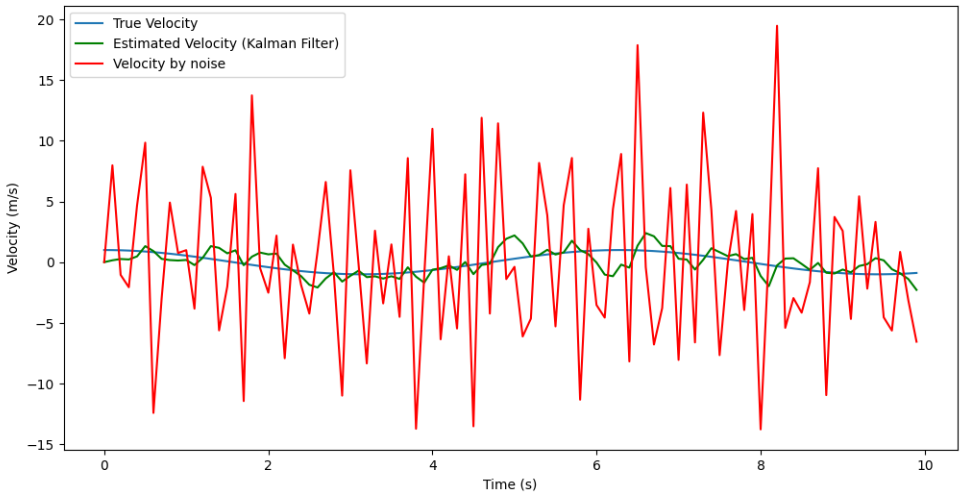
$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

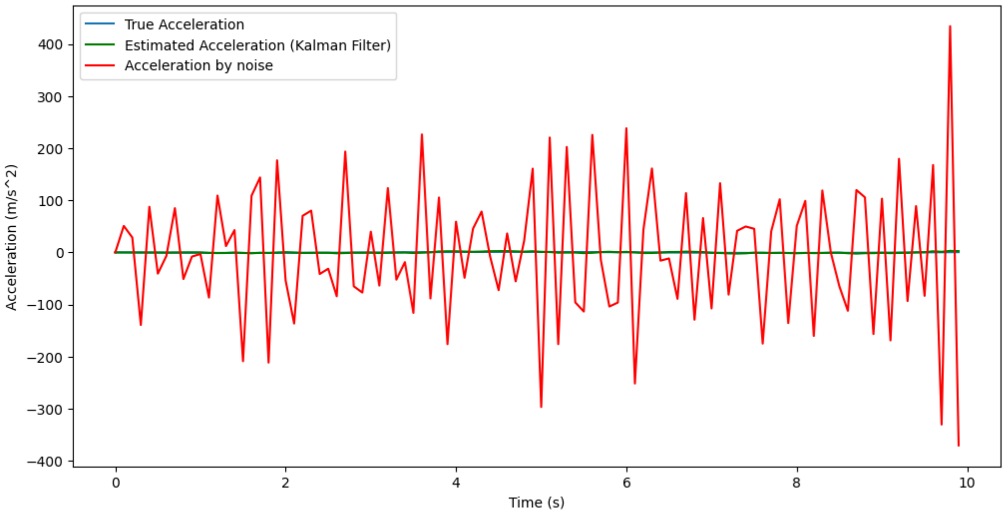
$$\Sigma_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

پس به شکل بالا تابع مورد نظر را مقدار دهی کردم .

در اندازه‌گیری‌های نویزی، موقعیت، سرعت و شتاب تخمینی را از حالت فیلتر شده استخراج می‌کنیم و نتایج را رسم می‌کنیم. نمودارهای به دست آمده نشان می دهد که فیلتر کالمن قادر است موقعیت واقعی، سرعت و شتاب سر فنر را علیرغم اندازه گیری های سنسور نویز ردیابی کند.







sd (انحراف معيار)	mse with filter	mse without filter
0.1	0.002	0.004
0.3	0.004	0.0048
0.6	0.05	0.12
0.9	0.22	0.38